

AD-A257 487



①



Proceedings of the
Second International Conference

**ALGEBRAIC
METHODOLOGY
AND
SOFTWARE
TECHNOLOGY**

May 22-25, 1991

S DTIC
ELECTE
NOV 19 1992 **D**
E

**The University of Iowa
Iowa City, Iowa**

188465

92-29843



DISTRIBUTION STATEMENT

Approved for public release
Distribution Unlimited



Mathematics well-applied illuminates rather than confuses

Second International Conference on Algebraic Methodology and Software Technology, AMAST¹

May 22-25, 1991, Iowa City, Iowa, USA

DTIC QUALITY INSPECTED 4

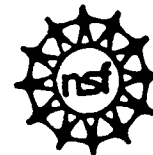
Organizing Committee:

General chairman: Maurice Nivat, University of Paris VII, France
Local chairman: Arthur Fleck, University of Iowa, Iowa City, IA, USA
Program chairman: Teodor Rus, University of Iowa, Iowa City, IA, USA
Finance chairman: Monagur Muralidharan, University of Iowa, IA, USA
Publicity chairman: Charles Rattray, University of Stirling, Scotland
Giuseppe Scollo, University of Twente, The Netherlands
Tomasz Müldner, Acadia University, Canada

Members:

Roland Backhouse, University of Groningen, The Netherlands
Michel Bidoit, University of Paris-South, France
Robert Constable, Cornell University, Ithaca, NY, USA
Hartmut Ehrig, Technical University of Berlin, West Germany
Marie-Claude Gaudel, University of Paris-South, France
Irene Guessarian, University of Paris VI, France
William S. Hatcher, Laval University, Quebec, Canada
Günter Hotz, University of Saarland, Saarbrücken, West Germany
Neil D. Jones, University of Copenhagen, Copenhagen, Denmark
William A. Kirk, University of Iowa, Iowa City, IA, USA
William F. Lawvere, State University of New York at Buffalo, NY, USA
Eugene Madison, University of Iowa, Iowa City, IA, USA
George Nelson, University of Iowa, Iowa City, IA, USA
Don Pigozzi, Iowa State University, Ames, IA, USA
Vaughan Pratt, Stanford University, California, USA
David Schmidt, Kansas State University, Manhattan, Kansas, USA
Ralph Wachter, Office of Naval Research, Arlington, Virginia, USA
Eric Wagner, IBM Thomas J. Watson Research Center, NY, USA

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	



¹This conference was sponsored by the Office of Naval Research, grant N00014-90-J-1836, National Science Foundation, grant CCR-90-14162, and The University of Iowa.

Organizing Committee's Message

The first AMAST meeting held two years ago was well received, and members of the organizing committee were encouraged by some who attended to propagate this forum. This was despite the fact that the goal for the purposes of the meeting was less than completely met. So we set forward these purposes again and enlist your support in the attempt to attain them more perfectly.

The goal of this conference is to foster algebraic methodology as a foundation for software technology and to show that this can lead to practical mathematical alternatives to ad hoc approaches commonly used in software development. We are particularly interested in showcasing software systems that have been developed through such an approach, and focusing on those conceptual developments that underlie the success of such accomplishments.

For the second AMAST meeting, the organizing committee received 91 submissions. From these submissions, available time forced us to construct a program containing 31 contributed talks. As you see from the program, the topics of these papers span a wide variety of software development issues, and we find this very encouraging. The program is rounded out by a distinguished collection of invited speakers. One last event we would like to call your attention to is the system demonstration session on Wednesday evening. While arrangements for this session are still being finalized at the time of this writing, this is where we hope to see algebraic methodology at work and we encourage you to take a look at what's going on there.

The organizing committee has also been exploring avenues for the regular publication of papers that contribute to the conference goals. In addition to these proceedings with brief papers, all authors presenting papers at the conference will have the opportunity to contribute full papers in a volume to be published by Springer-Verlag/London. Also, all conference participants are invited to submit papers for a special issue of the journal Theoretical Computer Science, and these submissions will go through the normal refereeing process. Finally, we are exploring the initiation of a new monograph series devoted to this area and expect to be able to announce final details soon.

The response of those of you with interests in this area has a significant effect on the outcome of all these efforts, so let your impressions be known, volunteer when you see a need, and we hope to see this area flourish.

Finally, we should ask where we go from here. The general agreement seems to be to strengthen and spread this conference even further. For that we need to move it around, such that the ideas set forth by it will catch the attention of the entire class of software designers. We can already state that there are good chances to organize AMAST3 in Enschede, The Netherlands, in May 1993. Since AMAST is unique in its goals, holding this conference at a variety of locations is probably very important now because the penetration of software in everyday life requires indeed practical approaches to develop it both soundly and efficiently. Promoting the idea of a (mathematical) algebraic methodology for software technology may be doubly beneficial: it will prevent the repetition of the errors generated by ad hoc development of software, and it will promote a field of science which has already had a strong impact on science and technology in general.

Conference Program

Tuesday, May 21-st, 7:30-9:30pm

Reception

Wednesday, May 22-nd, 8:00-8:30

Registration

Wednesday 22-nd, 8:30-9:00

Welcome: Hunter R. Rawlings III,

President of the University of Iowa.

Opening remarks: Nivat, M., Conference Chair.

Wednesday 22-nd, 9:00-12:30: Session.1

Algebraic Logic for Software Reusability

Chair: Rattray, C.M.I., University of Stirling.

♣ 9:00-10:00 Invited talk: *Event Spaces and Their Linear Logic*, Pratt, V., Stanford University.

♣ 10:00-10:30 Astesiano, E., Giovani, A., Morando, F., Reggio, G, *Algebraic Specification at Work*.

10:30-11:00 Coffee break

♣ 11:00-11:30 Parisi-Presice, F., *On the Reusability of Specifications and Implementations*.

♣ 11:30-12:00 Eichmann, D., *Selecting Reusable Components Using Algebraic Specifications*.

♣ 12:00-12:30 Hirst, G.S., Dinesh, T.B., *The Combination of Specifications and the Induced Relations in Object Oriented Programs*.

12:30-1:30 Lunch

Wednesday 22-nd, 1:30-5:30 Session.2:

Algebraic Design of Parallel Systems

Chair: Hotz, G., University of Saarland.

♣ 1:30-2:30 Invited Talk: *MEC: A System for Constructing and Analysing Transition Systems*, Arnold, A., University of Bordeaux.

♣ 2:30-3:00 Kaplan, S., Deutsch, G., *Real-Time Process Specification*.

♣ 3:00-3:30 Janicki, R., Koutny, M., *Structure of Concurrency*.

3:30-4:00 Coffee break

♣ 4:00-4:30 Inverardi, P., Nesi, M., *On Rewriting Behavioural Semantics in Process Algebra*.

♣ 4:30-5:00 Zamfir-Bleyberg, M., *Modeling Concurrency with AND/OR Algebraic Theories*.

♣ 5:00-5:30 Cornell, A., *Type Consistency Checking for Concurrent Independent Systems*.

7:30 System demonstrations

Thursday 23-rd, 8:30-12:30 Session.3:

Specification of Software Systems

Chair: Wagner, E., IBM Watson Res. Center.

♣ 8:30-9:30 Invited talk: *Theory of Algebraic Module Specification including Behavioural Semantics, Constraints, and Aspects of Generalized Morphisms*, by Ehrig, H., Baldamus, M., Cornelius, F., Orejas, F., presented by Ehrig, H., Technical University of Berlin.

♣ 9:30-10:00 Schobbens, P-Y., *Clean Algebraic Exceptions with Implicit Propagation*.

10:00-10:30 Coffee break

♣ 10:30-11:00 Fey. W., *ACT TWO: An Algebraic Module Specification and Interconnection Language*.

♣ 11:00-11:30 Talcott, C., *Towards a Theory of Binding Structures*.

♣ 11:30-12:30 Invited talk: *Proving Correctness of Algebraically Specified Software: Modularity and Observability Issues*, Bidoit, M., LIENS, Ecole Normale Supérieure, Paris.

12:30-1:30 Lunch

Thursday 23-rd, 1:30-4:30 Session.4:

Construction of System Software

Chair: Schmidt, D., Kansas State University.

♣ 1:30-2:30 Invited Talk: *A Formal Approach to Software Testing*, by Bernot, G., Gaudel, M-C., Bruno Marre, B., presented by Gaudel, M-C. University of Paris-South.

♣ 2:30-3:00 Hussmann, H., *A Case Study Towards Algebraic Verification of Code Generation.*

♣ 3:00-3:30 Knaack, J., *A Two Level Scanning Algorithm.*

♣ 3:30-4:00 van der Meulen, E.A. *Deriving Incremental Implementations from Algebraic Specifications.*

♣ 4:00-4:30 Cai, J., Paige, R., *Languages Polynomial in the Input Plus Output.*

4:30 - Riverboat Trip

Friday 24-th, 8:30-12:30 Session.5:

Reasoning about Software Development

Chair: Hatcher, W.S., Laval University.

♣ 8:30-9:30 Invited talk: *Polynomial Relators*, by Backhouse, R.C., de Bruin, P.J., Hoogendijk, P., Malcolm, G., Voermans, E., van der Woude, J., presented by Backhouse, R., Eindhoven University of Technology.

♣ 9:30-10:00 Breazu-Tannen, V., Subrahmanyam, R., *On Adding Algebraic Theories with Induction to Typed Lambda Calculi.*

10:00-10:30 Coffee break

♣ 10:30-11:00 Smith, S.F., *Extracting Recursive Programs in Type Theory.*

♣ 11:00-11:30 Zhang, H., Guha, A., Hua, X., *Using Algebraic Specifications in Floyd-Hoare Assertions.*

♣ 11:30-12:00 Kounalis, E., Rusinovitch, M., *Completion Procedures for Theories with Equality.*

♣ 12:00-12:30 Thomas, M., Watson, P. *Solving Divergences in Knuth-Bendix Completion by Enriching Signatures.*

12:30-1:30 Lunch

Friday 24-th, 1:30-6:00 Session.6:

Tools for Software Development

Chair: Mosses, P., Aarhus University.

♣ 1:30-2:30 Invited talk: *Efficient Algebraic Operations on Programs*, Jones, N.D., University of Copenhagen.

♣ 2:30-3:00 Karlsen, E.W., Krieg-Brückner, B., Traynor, O., de la Cruz, P., *Uniform (Meta-) Development in the PROSPECTRA Methodology and System.*

♣ 3:00-3:30 Eertink, H., *Tools for Algebraic Distributed System Design.*

3:30-4:00 Coffee break

♣ 4:00-4:30 Rus, D., *Using Lie Algebras for Dexterous Manipulations.*

♣ 4:30-5:00 Srinivas, Y.V., *Pattern-Matching: A Sheaf-Theoretic Approach.*

♣ 5:00-5:30 Bert, D., Lafontaine, C., *Integration of Semantical Verification Conditions in a Specification Language Definition.*

♣ 5:30-6:00 Lano, K., Haughton, H., *An Algebraic Semantics for the Specification Language Z++.*

7:30 Banquet

Saturday, 25-th, 9:00-1:00 Session.7:

Algebraic Software Technology

Chair: D. Pigozzi, Iowa State University.

♣ 9:00-10:00 Invited talk: *About Algebras, Fixpoints, and Semantics*, Guessarian, I., University of Paris VI.

♣ 10:00-10:30 Marciano, R., *Algebraic Construction of Program Dependence Graphs.*

10:30-11:00 Coffee break

♣ 11:00-11:30 Ramalingam, G., Reps, T., *Modification Algebras.*

♣ 11:30-12:00 Qin, H., Lewis, P., *Decomposition of Finite State Machines under Isomorphic and Bisimulation Equivalence.*

♣ 12:00-12:30 Antimirov, V.M., Melnikova, H.V., *Abstract Algebraic Implementations of Order-Sorted Specifications.*

12:30-1:00 Concluding Remarks

Referees

R. Alderden	V. Ambriola	M. Anderson
A. Arnold	P.R.J. Asveld	J. Baeten
V. Breazu-Tannen	A. Belinfante	B. Botma
E. Brinksma	S. Bruell	L. Cardelli
J. Chou	A. Ciepielewski	W. P. Coleman
P. Degano	T.B. Dinesh	M. Dodani
H. Eertink	H. Ehrig	D. Eichmann
P. van Eijk	D. Epley	A. Fleck
S. Ghosh	R. Gorrieri	X. Hua
H. Hussmann	D. Ionescu	R. Janicki
W. Janssen	D. Jones	P. Kars
J. Kearney	R. E. Kent	M. Koutny
H. Kremer	R. Langerak	P. Lauer
G. T. Leavens	C. Lengauer	L. Logrippo
V. Manca	R. Marciano	M. Mislove
E. Moggi	C. Montangero	P. S. Mulry
M. Muralidharan	J. O'Donnell	R. Oehmke
H. Oguztuzun	M. Oudshoorn	D. Pigozzi
L. F. Pires	M. Poel	V. Pratt
C. M. I. Rattray	A. Rensink	D. Rine
T. Rus	A. Salibra	D. Schmidt
J. Schot	A. Schoute	G. Scollo
P. Sestoft	M. van Sinderen	K. Slonneger
C. Talcott	M. Thomas	J. Tretmans
J. Urban	P. Watson	I. Widya
M. Zamfir-Bleyberg	H. Zhang	J. Zucker

Event Spaces and Their Linear Logic

Vaughan Pratt
Computer Sci. Dept., Stanford, CA 94305, USA
pratt@cs.stanford.edu

Abstract

Boolean logic treats disjunction and conjunction symmetrically and algebraically. The corresponding operations for computation are respectively nondeterminism (choice) and concurrency. Petri nets treat these symmetrically but not algebraically, while event structures treat them algebraically but not symmetrically.

Here we achieve both via the notion of an event space as a poset with all nonempty joins representing concurrences and a top representing the unreachable event. The symmetry is with the dual notion of state space, a poset with all nonempty meets representing choices and a bottom representing the start state. The algebra is that of a parallel programming language expanded to the language of full linear logic, Girard's axiomatization of which is satisfied by the event space interpretation of this language.

Event spaces resemble vector spaces in distinguishing tensor product from direct product and in being isomorphic to their double dual, but differ from them in distinguishing direct product from direct sum and tensor product from tensor sum.

This work was supported by the National Science Foundation under grant number CCR-8814921.

1 Introduction

A basic problem in the theory of concurrent computation is to define and reconcile nondeterminism and concurrency, respectively the disjunction and conjunction of behavior. Now in classical logic disjunction and conjunction have an appealing algebraic definition: we take them to be respectively the join $x \vee y$ and meet $x \wedge y$ of a distributive lattice, a concept axiomatizable with finitely many equations such as $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.

Join and meet are completely symmetric: the order dual of a distributive lattice, obtained by interpreting \leq as \geq , is still a distributive lattice with join and meet interchanged. When the lattice has a complement operation, as in a Boolean algebra, this duality is explicitly expressed in the language via De Morgan's laws, $(x \vee y)' = x' \wedge y'$ and $(x \wedge y)' = x' \vee y'$.

To date there has been no model of concurrency that combines algebra and symmetry the way Boolean algebra does for logic. The places and transitions of Petri nets are symmetric and act respectively as disjunction and conjunction. However there is no algebra of Petri nets whose operations correspond to respectively nondeterministic choice and concurrence of nets while preserving that symmetry. Conversely Winskel's event structures and prime algebraic domains [Win86] have an attractive algebraic theory but are not at all symmetric.

In this paper we give an algebraic model of computation that treats choice and concurrency symmetrically. It subsumes event structures with a minimum of machinery, while its algebra constitutes a parallel programming language whose syntax and axiomatics are most succinctly described as being those of full linear logic.

Our starting point is Winskel's notion of event structure, based on Birkhoff's duality of posets and distributive lattices [Bir33], see also [Sto36, Sto37, Pri70]. This duality associates to each poset $S = (X, \leq)$ the distributive lattice $2^{S^{op}}$ consisting of the order ideals of S , and to each automaton $A = (X, \vee, \wedge)$ the schedule $2^{A^{op}}$ consisting of the lattice ideals of A , in such a way that A is isomorphic to $2^{S^{op}}$ if and only if S is isomorphic to $2^{A^{op}}$. (We offer a brief tutorial on this duality at the end of this paper and a longer one elsewhere [Pra91].)

In the context of computation this duality has a natural interpretation in terms of schedules and automata, whose elements denote respectively events and states. This interpretation was found by Nielsen, Plotkin and Winskel [NPW81], who then extended the duality to incorporate a symmetric irreflexive binary *conflict* relation $\#$, with $x\#y$ forbidding the occurrence of x together with y . They required conflict to be *persistent*: if $x\#y$ and $y \leq z$ then $x\#z$. This extension, which they called an *event structure*, was subsequently developed much further by Winskel [Win86]. In the presence of conflict the automaton dual to $(X, \leq, \#)$ is the subposet of the distributive lattice dual to (X, \leq) consisting of the conflict-free states, those not containing both x and y when $x\#y$. A more general notion of conflict removes the binary restriction on conflict, allowing a set of events to be in conflict even though no proper subsets of that set are in conflict. The dual automaton is called *coherent* just when conflict remains binary.

Now the automaton dual to an event structure may not be a distributive lattice, due to conflict having deleted the top and other nonempty joins. However all nonempty meets are retained. Figure 2 below gives examples of such automata.

With this in mind let us take the existence of nonempty meets and the empty join as *definitive* of the kind of automaton we want. Specifically we define a *state space* to be a nonempty poset with all nonempty meets (including infinite meets). We provide an explicit constant symbol q_0 for the universal meet or empty join or bottom. A *map* of state spaces is a function between state spaces preserving nonempty meets and q_0 .

We then define the obvious dual notion to state space, which we shall call an *event space*. We define this to be a nonempty poset with all nonempty joins (including infinite joins), and having a constant symbol ∞ for the universal join or empty meet or top which will serve as the permanently deferred or impossible event. A *map* of event spaces is a function between event spaces preserving nonempty joins and ∞ . (Ordinarily we would call $q_0 \perp$ and $\infty \top$, but we have here reserved these symbols for respectively the two-state state space and the two-event event space.)

Section 2 treats the theory of a single event space, whose individuals are its events. There we find that event spaces subsume event structures, by permitting an arbitrary set of events to be in conflict. But whereas event structures schedule only atomic events, event spaces may schedule compound events.

Section 3 treats the theory of a calculus of event spaces under operations of a parallel programming language that turn out also to be the operations of linear logic. Their event space interpretation turns out to satisfy Girard's axiomatization of linear logic perfectly.

The duality of event and state spaces is more than just a matter of interchanging meet and join in

their respective definitions. There is a specific bijection between event spaces and state spaces¹ which has a strikingly simple description. The event space A^\perp dual to a given state space A is constructed from A by removing q_0 from the bottom and reinstalling it as ∞ at the top. When q_0 is removed some meets disappear; it is a theorem that we shall prove later that when it is reinstalled at the top all nonempty joins absent from A now appear. Performing the inverse operation on any event space, namely removing top and reinstalling it as bottom, similarly yields the dual state space, removing some joins and supplying all missing nonempty meets.

This leads to the even more striking conclusion that in this framework, with the exception of the initial state q_0 and the unattainable event ∞ , *individual events and individual states are the same notion*. Every state can be viewed as an event and vice versa.

The difference emerges only when we consider events or states in combination with each other. States combine via meet while events combine via join. The meet of a set of states constitutes the initial state for that set, while the join of a set of events amounts to their union, collecting all the events of the set into one compound event.

Event spaces behave very much like vector spaces. In particular they admit linear transformations: the set $b \rightarrow a$ of linear transformations from event space b to event space a itself forms an event space. (We denote both event spaces and state spaces by lower case variables a, b, c when treating their “external” algebra.) There is also a direct product $a \times b$, a tensor product $a \otimes b$, and a dual space $a^\perp = a \rightarrow \perp$ satisfying $a^{\perp\perp} = a$ analogous to the vector space V^* dual to V .

However event spaces differ from vector spaces in three ways. First, $V^{**} = V$ holds only for finite dimensional vector spaces V , whereas $a^{\perp\perp} = a$ holds for all event space. Second, the default notion of distance between two vectors in a vector space is their difference, imposing an inflexible triangle equality and requiring the addition of a separately defined norm to relax this to a more flexible triangle inequality. In an event space however, taking distance to be the truth of $x \leq y$ yields an inherently flexible triangle inequality, namely reflexivity. (We develop this relationship between posets and metric spaces in concurrency modeling considerably further elsewhere [CCMP89].) Third, sum and product of vector spaces degenerate to the same operation, and similarly for tensor sum and tensor product, whereas for event spaces these are all distinct operations, respectively concurrence and choice, and cointeraction and interaction.

The above account obtains event spaces as dual to state spaces, which in turn are a natural abstract formulation of the dual of event structures. However event spaces were not actually found in this way. Instead they arose while contemplating the duality of bipointed sets and cubical sets (as top-and-bottom-less Boolean algebras), and the self-duality of complete semilattices (posets with all joins). The *modus operandi* of the former, in combination with the delicate balance of the latter, prompted us to explore the consequences of small perturbations of the latter. It is remarkable that so small a change as interchanging top and bottom in complete semilattices could have such a beneficial impact both inside (subsumption of event structures) and out (our calculus of event spaces).

¹We are here regarding event spaces and state spaces as being defined only up to isomorphism, which is to say that the bijection is really between isomorphism classes of event spaces and isomorphism classes of state spaces. In terms of categories we have only an equivalence of categories, not an isomorphism.

2 Definition and Examples

An *event space* is a nonempty partially ordered set $S = (X, \leq)$ such that every nonempty subset $Y \subseteq X$ has a join or least upper bound $\bigvee Y$ in X . The join $\bigvee X$ of the whole set must be the top element, which we denote ∞ .

The following Hasse diagrams² depict the nine event spaces having up to four points (events).

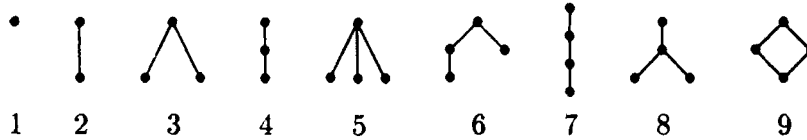


Figure 1. Event Spaces with at most 4 events.

A *state space* is the dual notion to event space, namely a nonempty partially ordered set $A = (X, \leq)$ such that every nonempty subset $Y \subseteq X$ has a meet or greatest lower bound $\bigwedge Y$ in X . The meet $\bigwedge X$ of the whole set must be the bottom element, which we denote q_0 (the traditional notation for start state) or $-\infty$.

The state space S^\perp dual to a particular event space S is obtained by deleting ∞ at the top and adjoining q_0 at the bottom. This operation is equivalent for Figures 1.1-1.7 to taking the order dual (turning the diagram upside down), and for 1.8 and 1.9 interchanging them as well as inverting them, as shown in Figure 2.

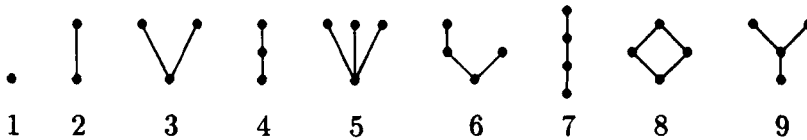


Figure 2. State Spaces Corresponding to Figure 1.

The number of event spaces (and hence state spaces) of each cardinality from 1 to 10 is 1, 1, 2, 5, 15, 53, 222, 1078, 5994, 37622. (It is straightforward to verify that this is also the number of lattices of each cardinality from 2 to 11: just add a bottom element to every event space, and delete it from every lattice.) The corresponding numbers for posets with 0 to 9 points are 1, 1, 2, 5, 16, 63, 318, 2045, 16999, 183231. I am grateful to Pat Lincoln, Vineet Gupta, and the problem solving class CS 204 for computing and verifying these figures.

Let us now illustrate the use of event spaces with some naive theories of the notion of family. Figure 3 depicts these theories while Figure 4 depicts the corresponding state spaces.

We begin with the free event space on a set $\{m, w, c\}$, figure 3(a). (We define “free” later.) Let us take 3(a) to denote the unconstrained evolution of a family consisting of a man, a woman, and a child. ∞ represents the event that will never happen, while the remaining events represent the entry

²A *Hasse diagram* depicts a partially ordered set as an undirected graph whose edges have an implicit upward orientation. Hence $x \leq y$ is represented by the existence of a path from point x leading upwards to point y . Since posets are reflexive there is an implicit self-loop at every vertex, and since they are transitive there is an implicit edge from x to z for every upward path $x \leq y \leq z$. Deleting y from the poset does not mean that $x \leq z$ no longer holds but rather that the implicit edge from x to z in the Hasse diagram now needs to be made explicit.

of a nonempty subset of those three individuals into the family. For example $m \vee w$ is the event where both the man and the woman enter the family. Equivalently it may be viewed as the state in which the man and the woman are both in the family.

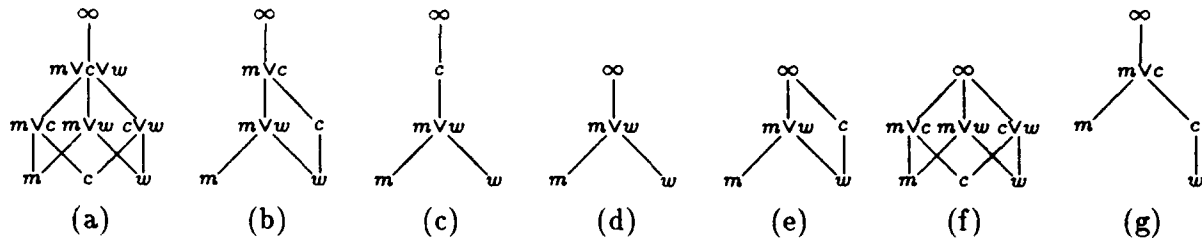


Figure 3. Event Space Representation of Some Theories of Families.

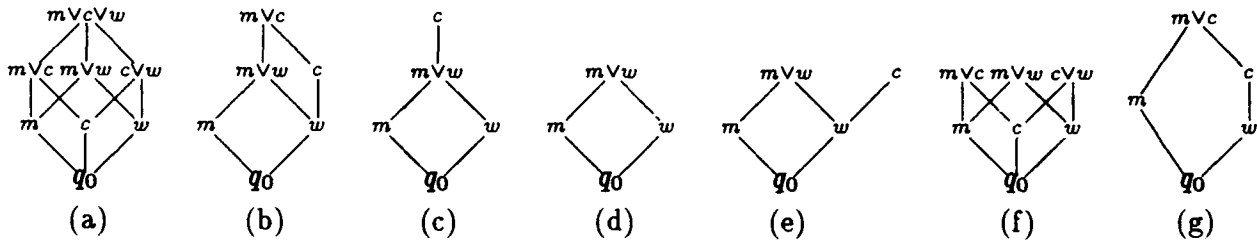


Figure 4. State Spaces Corresponding to Figure 3

We regard the man and woman as each having entered the family when each has made up their mind to do so. When both have made up their mind we have the *concurrent* event $m \vee w$, namely an event x representable as the join of two incomparable events. Let us call a nonconcurrent event other than ∞ *atomic*. In Figure 1, ∞ is concurrent in 3, 5, 6, and 9 while the only non- ∞ concurrent event is the one in 8 just below ∞ . In Figure 3 the concurrent events are those with \vee in their label, along with ∞ in 3(e) and 3(f).

One naturally supposes that one of m or w must have made up their mind to join the family no earlier than the other. This would imply that $m \vee w$ would be equal to one of m or w , or to both if they decided simultaneously. That the events m , w , and $m \vee w$ are all distinct shows that we have not made this supposition. In this naive account of the family we shall identify $m \vee w$ with marriage, which we could take to be a protocol for establishing common knowledge of the respective decisions of the man and the woman to so unite ("Do you take this...?"). As such it constitutes from at least a legal viewpoint the earliest moment at which the concurrency of m and w is observable. This is at once an event and a state.

The process in 3(a) is very general, allowing for the possibility of an unrelated man, woman, and already born child conspiring to form a family of three, with the child being adopted. Events $m \vee c$ and $c \vee w$ denote the mutual agreement of those pairs, which along with the agreement $m \vee w$ we are referring to as marriage are all assumed to be prerequisites to the formation of the whole family, namely the event $m \vee c \vee w$.

Let us now rule out adoption and consider henceforth only children born into a family. Call this requirement *motherhood*, a consequence of which is $w \leq c$.

Application of motherhood to 3(a) yields 3(b), removing some concurrency but not all: we still have the concurrency between the marriage $m \vee w$ and the child c inferrable from the parallelogram. We

infer a potential race condition, as in the closing scene of *Irma La Douce* in which the marriage is pronounced only seconds before the child is born.

We may describe the passage from 3(a) to 3(b) as either the result of deleting c and $m \vee c$ from (a) and renaming $c \vee w$ to c and $m \vee c \vee w$ to $m \vee c$, or as the result of identifying $c \vee w$ with c and $m \vee c \vee w$ with $m \vee c$ in (a). The former views (b) as a *subspace* of (a), the latter as a *quotient space*.

What makes the latter view particularly attractive is that it can be identified with an equation, namely $c \vee w = c$, synonymous with $w \leq c$. Let us fix this idea with some more examples.

The concurrency in 3(b) can be removed by specifying whether or not the child is born in wedlock, respectively 3(c) and 3(g). (Distinguish this requirement from that of the man being the biological father of the child, which is outside the scope of this example.) For now consider just 3(c). (This does describe Irma's marriage, but had the timing been sufficiently close as to raise doubts or disagreement, the observers might feel obliged to report having *observed* only 3(b) even while agreeing among themselves that one of 3(c) or 3(g) (discussed below) must surely have taken place in God's eyes.)

As with the relationship of 3(b) to 3(a) we can regard 3(c) as either a subspace or a quotient of 3(b). The latter view is captured by the equation $m \vee c = c$ or $m \leq c$, a child may only be born into a family already equipped with a man. In combination with motherhood this becomes $m \vee c \vee w = c$, or $m \vee w \leq c$.

Suppose now that the man requires that the marriage if any be childless; this gives rise to 3(d). While it is clear how 3(d) can be seen as a subspace of 3(c), it is not clear how it could be a quotient. This is where the inaccessibility of ∞ enters: we may regard 3(d) as the result of identifying c and ∞ in 3(c), hence forbidding c not by its absence but by its position at the end of time. This is not of itself a conflict in our sense, but it leads naturally into that notion.

Conflict. The event ∞ is the last event, which we view as never happening. When $\bigvee Y = \infty$ for any set Y of events not containing ∞ we say that Y is *in conflict*. The examples in Figure 1 that contain sets of events in conflict are 3, 5, 6, and 9, while those in Figure 3 are (e) and (f). The corresponding automata in Figure 4 have more than one maximal element (final state).

Note that it is possible for a large set of events to be in conflict without any subset being in conflict. An example of this appears as Figure 3(f), constructed from figure 3(a) by identifying $m \vee w \vee c$ and ∞ . This identification puts the set $\{m, w, c\}$ in conflict without putting any of its subsets in conflict. Concretely this means that any two out of the three pairs $m \vee c$, $m \vee w$, and $c \vee w$ are permitted, but nothing larger. Thus this schedule amounts to a program making a three-way decision determining which of the three is eventually left out.

Conflict is also meaningful in continuous situations. Consider the point $x = 1 - 1/(1 + t)$ moving as a function of time t ranging over the nonnegative reals. The point starts at 0 and moves towards 1 without ever reaching it. Its behavior is therefore modeled by the event space $[0, 1]$ of reals with its standard order, with \bigvee interpreted as sup and ∞ as 1. Any subset of $[0, 1]$ with sup 1, e.g. $[0, 1)$, is in conflict, that is, at no time can the set of points visited thus far have sup 1.

On the other hand the point $x = t$ where t ranges over $[0, 1]$ reaches $x = 1$ in unit time. A suitable event space describing this situation consists of the unit interval as before but with the integer 2 adjoined to take over the role of ∞ from 1. Now no subset of $[0, 1]$ is in conflict.

In this continuous example conflict was used only to encode inaccessibility of a limit, not an actual choice. Conflict in a finite set of events on the other hand necessarily entails a choice. Consider

imposing the childless-marriage requirement on Figure 3(b), in the absence of wedlock. This yields 3(e), which permits the woman to have a child out of wedlock. This presents her with a *conflict*: she can have either a child or a man in her family, but not both.

Event spaces do just as well as event structures in scheduling atomic events and specifying which sets are in conflict. However event spaces can also schedule concurrent events, which event structures cannot. For example we may write $c \leq m \vee w$ to indicate that the child is born out of wedlock, or more precisely that it enters the family before the marriage.

Applying this constraint to the free event space 3(b) yields the non-free space 3(g). Note that the corresponding state space 4(g) is a well-known nondistributive lattice. We will explain free spaces in the next section.

Example 3(g) expresses marriage to a woman with a child born out of wedlock and amounts to Figure 3(b) plus the condition $c \leq m \vee w$, yielding Figure 3(g). Here the man marries the unit consisting of the woman and her child, which we take to be linearly ordered (if discretely ordered we obtain another non-free example). The essential feature of 3(g) is that there is only one event following m , c , and w , namely $m \vee c \vee w$, or $m \vee c$ given that $w \leq c$ here. That is, we wish to dispense with the distinctions between the two-element subsets of $\{m, w, c\}$ making up 3(b) (or 3(a) in the absence of $w \leq c$), and simply say that when m is marrying a woman with a child there is only one resulting concurrent event. For when we draw such distinctions as in 3(b) then we admit the possibility of c 's entering the family later and hence legitimately instead of as an instantaneous consequence of m 's marriage to w .

3 Event Maps and Free Event Spaces

Event maps. Given event spaces $S = (X_S, \vee_S, \infty_S)$ and $T = (X_T, \vee_T, \infty_T)$, an *event map* $f : S \rightarrow T$ is a homomorphism of event spaces. That is, it is a function $f : X_S \rightarrow X_T$ satisfying $f(\vee_S Y) = \vee_T f(Y)$ for all nonempty $Y \subseteq X_S$, and $f(\infty_S) = \infty_T$.

This is in contrast to a monotone function or poset map, which is a function $f : (X_S, \leq_S) \rightarrow (X_T, \leq_T)$ such that if $x \leq_S y$ then $f(x) \leq_T f(y)$. Now if $x \leq_S y$ then $x \vee_S y = y$ so $f(x) \leq_T f(x) \vee_T f(y) = f(x \vee_T y) = f(y)$. Hence every event map is a poset map.

We do not however have the converse. In Figure 1, the map from example 1 to example 2 that takes the single element of 1 to the lower element of 2 gives an example of a poset map that is not an event map because it fails $f(\infty_1) = \infty_2$. Another such example is the map from 3 to 2 that takes the two lower elements of 3, call them x and y , to the lower element of 2, for then we have $f(\infty_3) = f(x \vee_3 y) = f(x) \vee_2 f(y) \neq \infty_2$.

Implicit in our definition of event map as a homomorphism of event spaces is the *signature* of an event space, that is, the "official" operations and constants. The signature consists of the operation \vee and the constant ∞ . An event map is a homomorphism of event spaces by virtue of preserving the operations and constants of the signature.

Note the more substantive role ∞ now plays. Previously ∞ was merely a synonym for $\vee X$. However the most we can infer from $f(\vee_S Y) = \vee_T f(Y)$ is that $f(\vee_S X_S) = \vee_T f(X_S)$. But $\vee_T f(X_S)$ need not be $\vee_T X_T$, witness the map f from example 1 to example 2 in Figure 1 taking the one element of 1 to the lower element of 2.

One use of event maps is to describe the process of adding events. An *injective* event map $f : S \rightarrow T$, one for which $f(x) = f(y)$ implies $x = y$, makes T the result, up to isomorphism, of adding events to S . The requirement $f(\infty_S) = \infty_T$ then means that every added event must precede ∞ . This confers on ∞ the status of an absolutely final event, one that no event can ever follow.

When $f(x) = \infty$ this means only that f has pushed x sufficiently far into the future as to be unable to distinguish it from ∞ . It does not lessen the absoluteness of ∞ itself, which can never be mapped to anything earlier than ∞ .

Free Event Spaces. We have seen that event spaces are everything that posets are and more. The event spaces that are not more are called the *free* event spaces. Those that are more are not free by virtue of being constrained by equations, either $\bigvee Y = \infty$ expressing conflict or $\bigvee Y = \bigvee Z$ scheduling compound events, e.g. $c \leq m \vee w$ in 3(g). Such constraints are not representable by order alone, in contrast say to $m \vee w \leq c$ in the free space 3(c) which is mere poset information, namely $m \leq c$ and $w \leq c$.

Associated with each event space $S = (X, \bigvee, \infty)$ is its *underlying* poset $U(S) = (X, \leq)$ where $x \leq y$ just when $x \vee y = y$. Another associated poset is its *compact* subposet, consisting of the *compact* elements of S , namely those elements $x \neq \infty$ such that for any nonempty $Y \subseteq X$, $x = \bigvee Y$ implies $x \in Y$. In the finite case the compact elements of S are just those other than ∞ with at most one edge leading up to it in the Hasse diagram of S . In the seven examples of Figure 3 these are in every case the elements labeled with just a letter, one of m , c , or w . Exercise: identify the 19 compact elements in Figure 1.

A *free* event space $F(P)$ on a poset P is an event space that behaves just like P with respect to labeling. More formally, for every event space S (serving as a source of labels) the set of event maps $f : F(P) \rightarrow S$ are in bijective correspondence with the set of monotone functions or poset maps $f' : P \rightarrow U(S)$, where f' is just the restriction of f to P . We may "find" P inside $F(P)$ as just its compact elements.

In Figure 1, event spaces 3, 5, 6, and 9 are not free because in each case ∞ is the join of the set of compact elements. Hence if we take $S = 2$, the 2-element event space with elements $0 \leq \infty$, and label the compact elements 0, then any extension of this labeling to an event map must map ∞ to both 0 and ∞ ; hence this poset map from P to $U(S)$ has no corresponding event space map from $F(P)$ to S , showing that the event space in question is not free. The remaining spaces of Figure 1 are free.

In Figure 3 the same argument shows that spaces (e) and (f) are not free. In space (g), still taking $S = 2$, label m, w, c respectively 0, 0, ∞ . Now $m \vee c$ is the join of both $\{m, w\}$ and $\{m, c\}$ and hence must be labeled both 0 and ∞ . The remaining spaces of Figure 3 are free.

This characterization of $F(P)$ as the event space equivalent to its compact subposet is not very constructive. An explicit construction of $F(P)$ given only P is as the set of nonempty order ideals³ of P , with \bigvee interpreted as union, together with a separate top element ∞ .

The correspondence between the maps from $F(P)$ to S and those from P to $U(S)$ yields two interesting maps. Taking $S = F(P)$, the identity event map from $F(P)$ to itself corresponds to a poset map called $\eta_P : P \rightarrow UF(P)$, the *embedding* of P in $F(P)$. Taking $P = U(S)$, the identity poset

³An *order ideal* of a poset (X, \leq) is a subset $Y \subseteq X$ such that if $x \leq y$ then $y \in Y$ implies $x \in Y$. The union of any set of nonempty order ideals is clearly a nonempty order ideal. An order ideal is *principal* when it has a top, equivalently when it is the set of elements less or equal to a single element, said to *generate* that ideal.

map from $U(S)$ to itself corresponds to an event map called $\varepsilon_S : FU(S) \rightarrow S$. If we think of the points of $FU(S)$ as terms whose variables are the points of S then ε_S is the *evaluation* map giving the value of each term.⁴

If ε_S is evaluation then its restriction to S , the variables of $FU(S)$, must be just the identity on S . This is expressed formally by the requirement that the composition of ε_S , more precisely of $U(\varepsilon_S) : UFU(S) \rightarrow U(S)$, with $\eta_{U(S)} : U(S) \rightarrow UFU(S)$ be the identity on $U(S)$. Less intuitive but equally clear formally, we require that the composition of $\varepsilon_{F(P)} : FUF(P) \rightarrow F(P)$ with $F(\eta_P) : F(P) \rightarrow FUF(P)$ (the image of η_P under F) be the identity on $F(P)$.

If P was obtained as the compact subposet of $F(P)$ then η_P is merely the corresponding inclusion. If in addition P happened to have all nonempty joins this would make it the underlying poset $U(S)$ of some event space S , in which case $\varepsilon_S : FU(S) \rightarrow S$ would be the identity on P and take ∞ to itself; the remaining elements of $F(P)$ are of the form $x = \bigvee_{F(P)} Y$ for $Y \subseteq P$ (exercise) and so are mapped by ε_S to $\bigvee_S Y$, within S .

If on the other hand we obtained $F(P)$ by explicit construction from P via order ideals then η_P must be given as part of the construction: it is the function taking each $x \in P$ to the principal order ideal generated by x . Either way η_P is injective, that is, it embeds P in $UF(P)$, allowing us to think of P as a subposet of $UF(P)$ whether or not it actually is.

Set-Free Event Spaces. Substituting “set” for “poset” everywhere in the definition of free event space yields the notion of free event space on a set instead of on a poset. We distinguish this notion of free via the term set-free. Of the five free event spaces in Figure 1, only 1, 2, and 8 are set-free, while in Figure 3 (b) and (c) are the two free event spaces that are not set-free. If instead of sets we had substituted event structures for posets we would arrive at the notion of free event space on an event structure. Figures 3(f) and 3(g) are the only counterexamples we have seen to free event spaces on an event structure, and 3(f) is only a counterexample if we restrict to coherent event structures, those with only a binary conflict relation $\#$.

Conflict-Free Event Spaces. We shall call those event spaces such that $\bigvee Y = \infty$ implies $\infty \in Y$ *conflict-free*. The corresponding notion of underlying object $U(a)$ is just that of forgetting the constant ∞ . The matching $F(a)$ adjoins ∞ at the top of a . Note that in the absence of ∞ it becomes possible to have an empty event space, the free event space on which is then the one-point space.

Conflict-free event spaces are as the name suggests free of conflicts.

4 A Calculus of Event Spaces

We turn now from the logic of events in a single event space to a calculus of event spaces. This calculus will resemble the algebra of natural numbers under addition $x + y$, multiplication xy , exponentiation x^y , and “negation” 0^x , and similarly that of propositions under disjunction $p \vee q$, conjunction $p \wedge q$, implication $q \rightarrow p$, and negation $\neg p = p \rightarrow 0$.

The corresponding operations for event spaces will be tensor sum $a \oplus b$, tensor product $a \otimes b$, implication $b \multimap a$, and negation a^\perp . However we will also be interested in the underlying poset $U(a)$

⁴Categorically speaking these are of course the components of an adjunction $F \dashv U$ with unit η and counit ε . However it is not necessary to know any category theory to understand these components and how they fit together.

of each event space a . We therefore expand this algebra with a parallel list of poset operations, namely direct sum $a + b$, direct product $a \times b$, and implication $b \Rightarrow a$. We use the same negation operation a^\perp for both event spaces and posets. The zeroary forms of \oplus , \otimes , $+$, and \times are the respective constants \perp , \top , 0 , and 1 .

This appears to call for two sorts of terms in this calculus, one for event spaces and one for posets. However we can take advantage of the fact that $F(P)$ "behaves like" P and work with $FU(a)$ rather than $U(a)$, so that all objects are officially event spaces even when trying to behave like posets. We write $FU(a)$ as $!a$. Since all other monotone (i.e. covariant) operations have duals, $!$ may as well too, so we abbreviate $(!(a^\perp))^\perp$ to $?a$.

These nine operations and four constants are exactly those of Girard's linear logic [Gir87], a connection we will return to in a later section. The operations $a + b$, $a \times b$, and $a \otimes b$ arise naturally as operations of a parallel programming language, also treated in its own section.

We now interpret these operations and constants for event spaces. We define the negation a^\perp of a as the order dual of all of a except ∞ , which remains at the top; we prove below that this yields an event space. (Applying this form of negation to Figure 1 yields not Figure 2 but rather its order dual; all that changes in Figure 1 is then that 8 and 9 are interchanged.) We have already seen the definitions of $!a$ and $?a$. And we define the constants 0 and 1 as both being the one-point event space, Figure 1.1, and \perp and \top as the two-point event space, Figure 1.2. (We distinguish 0 from 1 and \perp from \top later using signed event spaces.)

We now define the three arithmetic operations for each of event spaces and posets, which we may tabulate thus.

	Implication via Maps	Product via Curry	Sum via De Morgan
Event Space Arithmetic :	$b \multimap a$	$a \otimes b$	$a \oplus b$
Poset Arithmetic :	$b \Rightarrow a$	$a \times b$	$a + b$

First we define column 1, the implications. We take $b \multimap a$ to be the poset of all event maps from b to a , ordered *pointwise*, that is $f \leq g$ just when $\forall x[f(x) \leq g(x)]$; we show shortly that this poset is in fact an event space. We take $b \Rightarrow a$ to be the poset of all poset maps from $U(b)$ to $U(a)$, which as we have seen corresponds to the poset of all event maps from $FU(b)$ to a , i.e. $b \Rightarrow a = !b \multimap a$.

Next we define column 2, the products. For event spaces the Curry principle or *s-m-n* theorem or residuation for event spaces is $(b \otimes c) \multimap a = c \multimap (b \multimap a)$, uniquely (up to isomorphism) defining the *tensor product* $a \otimes b$. For posets this principle becomes $(b \times c) \Rightarrow a = c \Rightarrow (b \Rightarrow a)$, uniquely (up to isomorphism) defining the *direct product* $a \times b$.

Lastly we define column 3, the sums. Define *tensor sum* $a \oplus b$ to be $(b^\perp \otimes a^\perp)^\perp$, the De Morgan dual of tensor product. Similarly define *direct sum* to be the De Morgan dual of direct product, $a + b = (a^\perp \times b^\perp)^\perp$.

Tensor product $a \otimes b$ may be defined alternatively as $(b \multimap a^\perp)^\perp$, via the reasoning $a \otimes b = ((a \otimes b) \multimap \perp)^\perp = (b \multimap (a \multimap \perp))^\perp = (b \multimap a^\perp)^\perp$. Direct product $a \times b$ may be defined alternatively as having an underlying set consisting of the Cartesian product of the underlying sets of a and b . Its top is (∞_a, ∞_b) while join is computed pointwise: the join of Y is $(\bigvee_a Y_a, \bigvee_b Y_b)$ where $Y_a = \{x \mid \exists y[(x, y) \in Y]\}$ and similarly for Y_b . It is readily seen that $a \times b$ so defined is an event space. While we have no better elementary definition of $a + b$ than as the De Morgan dual of $a \times b$, categorically speaking $a + b$ and $a \times b$ are naturally described as respectively coproduct and product.

This phenomenon of having two parallel lists of arithmetical operations is not unusual, occurring for example with relation algebras [JT48], vector spaces, and relevant logic [Dun84]. Concentrating on both rows of column 2, products, we find respectively a structured and unstructured product as follows for each of these examples. For relations we have composition $R; S$ and intersection $R \cap S$. For vector spaces we have tensor product $U \otimes V$ and direct product $U \times V$. For relevant logic we have cotenability $a \circ b$ and conjunction $a \wedge b$.

In each case we sometimes want to do arithmetic with the structure present and sometimes without it. The latter arithmetic is performed in effect on the set of elements of the structure, i.e. its underlying set, which is how we tend to work with event spaces from a set theory perspective rather than category theoretically. (Thus this is how to do set theory within category theory.) In our calculus we have chosen the underlying poset instead, although the axioms of the calculus would look about the same had we chosen the underlying set. In either case the underlying objects form a cartesian closed category (tensor product is direct product), in contrast to event spaces for which tensor product as interaction is distinct from direct product as choice.

Theorem 1 *The poset a^\perp constructed from a as above by inverting (taking the order dual of) all of a but ∞_a is an event space.*

Proof: The construction automatically equips a^\perp with a top, namely ∞ . It remains to show that the join of any nonempty subset Y of a^\perp exists in a^\perp . The following closely parallels the argument that a complete semilattice is a complete lattice.

If ∞ is in Y then $\bigvee Y = \infty$. Otherwise consider the set $\mathcal{L}Y$ of lower bounds on Y in a . If $\mathcal{L}Y$ is empty then the only upper bound on Y in a^\perp will be ∞ , which is then the least upper bound on Y , the desired join $\bigvee Y$. If $\mathcal{L}Y$ is nonempty then it has a join j in a . Now every element of Y is an upper bound on $\mathcal{L}Y$, whence j as the least upper bound must be a lower bound on Y . Hence it is the greatest lower bound on Y . But then it is the least upper bound on Y in a^\perp . ■

Although we defined negation explicitly in elementary terms, there is an equivalent algebraic definition of a^\perp .

Theorem 2 $a^\perp = a \multimap \perp$.

Proof: We identify each event map $f : a \multimap \perp$ with its kernel $f^{-1}(0)$ (0 being the lower element of \perp), which we call an *event space ideal*. The empty set is an event space ideal, and corresponds to the top map, which is therefore ∞ in $a \multimap \perp$. Since event maps preserve joins, every nonempty event space ideal must contain its join. But this makes it a principal order ideal, of which there is exactly one per element of a . Conversely every principal order ideal determines a map of $a \multimap \perp$ except the one generated by ∞_a , which corresponds to a map taking ∞ to 0, which is not an event map. We may therefore put the non- ∞ elements of a in one-one correspondence with the non-top maps of $a \multimap \perp$; let f_x denote the map corresponding to x . It is clear that $x \leq y$ if and only if $f_y \leq f_x$. Hence the poset of nontop maps of $a \multimap \perp$ is the order dual of the poset of non- ∞ elements of a , while the top map stays at the top. ■

Theorem 3 *The poset $b \multimap a$ consisting of all event maps from b to a , ordered pointwise, is an event space.*

Proof: The top event map is of course just the constantly top map, satisfying $f(x) = \infty_a$ for all events $x \in b$.

Given any nonempty set F of event maps $f : b \rightarrow a$, take its join $\bigvee F$ to be the function $g : b \rightarrow a$ defined as $g(x) = \bigvee_{f \in F} f(x)$ for each event x in b . It remains to show that g is an event map.

Now $g(\infty_b) = \bigvee_{f \in F} f(\infty_b) = \bigvee \infty_a = \infty_a$, whence g preserves ∞ . That g preserves nonempty joins follows thus.

$$\begin{aligned}
 g\left(\bigvee_{y \in Y} y\right) &= \bigvee_{f \in F} f\left(\bigvee_{y \in Y} y\right) \\
 &= \bigvee_{f \in F} \bigvee_{y \in Y} f(y) \\
 &= \bigvee_{y \in Y} \bigvee_{f \in F} f(y) \\
 &= \bigvee_{y \in Y} g(y).
 \end{aligned}$$

■

5 Signed Event Spaces

As things stand, if a is an event space so is a^\perp . However it is very natural to view a^\perp as the state space dual to a , not by inverting the non- ∞ portion of a but instead by simply moving ∞ from top to bottom and renaming it q_0 , as we saw in Figures 2 and 4.

From this perspective the non- ∞ events turn into states *without changing the partial order*. What does change are the joins and top, which are replaced by meets and a bottom. This is easily seen when it is noticed that this new notion of a^\perp is no more than the order dual of the old notion of a^\perp as an event space. Since the latter had the nonempty joins and top of an event space, the former as its order dual must have all nonempty meets and a bottom.

We define a *signed* event space to be a pair (a, s) where a is an event space and $s \in \{0, 1\}$ is a “sign bit” giving the direction of time. When the sign bit is zero the associated event space a is a schedule, its points are events, and $x \leq y$ means that x occurs before y . When the sign bit is one, a becomes a state space or automaton whose points are now states, and the temporal relationship that we write as $x \leq y$ in the event space interpretation of a is now to be viewed as the transition $y \geq x$ from y to x .

We extend the calculus of event spaces to a calculus of signed event spaces via the usual Boolean calculus of zero and one. Sum, product, and exponential, whether ordinary (poset) or tensor (event space), are respectively disjunction, conjunction, and implication. Negation exchanges 0 and 1. The comonad $!a$ has nothing to forget and so acts as the identity. We call the resulting category **2**.

The category **Sev** of signed event spaces is then the product $\mathbf{Ev} \times \mathbf{2}$.

In **Sev**, negation turns event spaces into state spaces and vice versa. The maps from an event space s to a state space a must be the states of the state space $s \multimap a$, since $0 \multimap 1 = 1$. This makes excellent

sense: the state of an event space as a whole is given by the states of each of its events. When $a = \top$, the automaton whose two states can be viewed as *not-done* and *done*, $s \multimap \top$ is the automaton whose states indicate which *atomic* states of s are done. But since s also includes its compound events the states of $s \multimap \top$ are actually just the events of s .

Now in **Ev** we have $0 = 1$ and $\perp = \top$. In **2** however we have $0 = \perp$ while $1 = \top$. Hence in **Ev** \times **2** all four constants are distinct. This removes the one objectionable degeneracy we are aware of in **Ev**.

The theory of signed event spaces can be seen to be the intersection of the theory of event spaces with the theory of 0 and 1 . Since the only significant isomorphisms in the difference that we are aware of are $0 = 1$ and $\perp = \top$, this intersection then accomplishes for us only that much.

Had we performed this intersection for the category **CSLat** of complete semilattices to yield the category of signed complete semilattices, or posets with all joins, we would have removed in addition the degeneracy of sum and product in **CSLat**. I do not know whether **CSLat** \times **2** contains any isomorphisms representable with such formulas that are not also present in **Ev** \times **2**. However if $!a$ were (the event space mimicking) the underlying set instead of the underlying poset of a we would have a distinction, since then $!a$ and $?a$ would be isomorphic complete atomic Boolean algebras, the set of atoms of $!a$ and the set of coatoms of $?a$ both being the underlying set of a . That degeneracy is not removed by the sign trick since it is present in **2** where both $!$ and $?$ are the identity.

Since the only difference between a schedule and its dual automaton is the location of the one bound, respectively at the end or the beginning, it is natural to ask, why not just put in both bounds and be done with the distinction altogether between schedules and automata?

One problem this unification creates is that $b \multimap a$, which is at the heart of our calculus, gains new maps since there is now an additional element in a to send elements of b to. This in turn throws off the nice balance of the calculus: b^\perp need have neither bound.

Indeed this is the principle behind the duality of bipointed sets, sets with two distinguished elements, and Boolean algebras without top or bottom. Contemplation of this duality, which Bill Lawvere suggested to me in a phone conversation as a simple construction of the theory of cubical sets, led me to the idea of evening things up by moving one or the other of the bounds over to the other side of the duality, and doing the same with one of the nonempty meets or the nonempty joins. When all the joins including bottom as the empty join are on the same side this yields complete semilattices, for which sum and product are the same operation up to isomorphism, and likewise tensor sum and tensor product, and there is no mechanism for representing conflict. The other way round yields event spaces, in which sum denotes concurrence while product denotes choice (whence they had better not be isomorphic), and conflict is now representable as the equality of top with a join.

Another change with such a unification is that the product or sum of a schedule with an automaton now has a quite different meaning. In **Sev** such a product orients the schedule opposite to the automaton so as to put their bounds at the same end before performing the addition or multiplication. In the unified view product is computed with the bounds at opposite ends. Whether this different meaning is better or worse is a very interesting question that clearly depends on why one cares about the product of an automaton with a schedule in the first place. If all we want is an internally consistent calculus, signed event spaces achieve this much. If for whatever reason we want temporal coherence in mixed product, some other approach is called for.

Partial Distributive Lattices. We have been investigating the possibility of a unification of schedules

and automata into a single much larger category that includes as well as most of the other basic structures encountered in the context of the Birkhoff-Stone duality, including sets, topological spaces, and Stone spaces, though excluding frames and locales which arise from a false dualizer, the two-point Sierpiński space. This is the category **PDL** of *partial distributive lattices*, having an abstract definition we dispense with here, but concretely representable to an as yet underdetermined extent as subsets of power sets. The power set on n elements has of course 1, 2, 4, 16, 256, ... elements starting at $n = -1$ (taking $2^{-1} = 0$), but these are grouped into respectively 1, 2, 4, 12, 70, ... isomorphism classes. Each class is a subclass of the larger ones, except for $n = 0$ which contains the one-point PDL which is both bottom and top, dual to the empty PDL and occurring nowhere else. The 70 classes at $n = 3$ partition as 1+3+5+11+20+16+9+4+1 by size from 0 (the empty subset) to 8 (the whole 3-cube) elements. The internal hom is obvious on reflection. Duality, defined by taking the complete two-element lattice as the dualizer in the usual way, is involutory for some but not all PDL's, e.g. the unit interval of reals with all meets and joins, and the 3-cube less one atom and its complementary coatom (found by V. Gupta). We hope to report further on properties and applications of this fascinating and potentially very useful category in a future paper.

Final states. We define a *final state* of an automaton to be the inf of a maximal chain of the automaton, necessarily a member of that chain. Maximal chains without such infs represent nonterminating computations. This uses the poset structure to code with an inf what Hoare has represented with \surd , and the absence of which we represented as the limbo state Λ in an early process logic paper [Pra79], coded here with an infinite chain lacking an inf.

6 Programming with Arithmetic

We give here the sense in which sum is concurrence, product is choice, and tensor product is interaction or orthocurrence [Pra86, CCMP89].

For schedules defined as posets, concurrence $P||Q$ can be defined simply as juxtaposition or poset coproduct $P + Q$, a point of view advocated by Grabowski [Gra81] and the author [Pra85, Pra86]. For event spaces that "mimic" a poset P , namely the free event space $F(P)$, coproduct remains the appropriate definition because F is a left adjoint and hence preserves (distributes over) coproducts. In particular the event space $F(P) + F(Q)$ mimicking $P + Q$ is $F(P + Q)$, which by this distributivity is $F(P) + F(Q)$, the coproduct of the spaces mimicking P and Q respectively.

Figure 3(b) for example is the concurrence of event m with the sequence $w \leq c$. As a poset this concurrence has just those three events, but as an event space it has besides ∞ the additional events $m \vee w$ and $m \vee c = m \vee c \vee w$. Figure 3(d) is the coproduct of m with w , while 3(a) is the coproduct of 3(d) with c .

Choice as product $a \times b$ is a more delicate notion. If we split up a as $\alpha; \infty$ and b as $\beta; \infty$ then

$$a \times b = \alpha \times \beta + \alpha \times \infty + \infty \times \beta + (\infty, \infty)$$

Theorem 4 *For conflict-free event spaces a, b , any two elements of $a \times b$ in conflict must be one from each of $\alpha \times \infty$ and $\infty \times \beta$.*

Proof: Consider (x, y) and (x', y') . Neither can be top, (∞, ∞) . Their join is $(x \vee x', y \vee y')$. Since a and b are conflict-free we must have either x and y' being ∞ , or x' and y , as claimed. ■

If we are given $a \times b$ unlabelled, with a and b conflict-free, even without the help of labels it is possible to identify two maximally conflict-free sets such that each element of one is in conflict with each element of the other. We may then infer that these must be $\alpha \times \infty$ and $\infty \times \beta$, which we may identify respectively with components a and b of the choice. We may think of the unconflicted portion $\alpha \times \beta$ as the portion of $a \times b$ responsible for making the choice of a or b .

The simplest example of such a choice is Figure 1.9, which is the product of two copies of Figure 1.2. The choice of which events to perform is between the two lower left events or the two lower right events, as the dual Figure 2.9 makes explicit. Since bottom is common to these we regard it as part of the decision making process. The two remaining events on the left and right, which together are in conflict, are then what we will have chosen to do. We were able to draw this distinction between decision making and resulting decision solely on the basis of the abstract structure of the event space, without reference to labels on events.

Interaction or othocurrence $a \otimes b$ is the notion of interacting particle systems such as colliding galaxies and a sequence of trains passing through a sequence of stations. Interaction is defined, and example applications given, in [Pra86] (where it is notated $a \times b$ because the coincidence $a \times b = a \otimes b$ in **Pos** led us to believe interaction was ordinary product) and in [CCMP89] by which time we had understood the distinction between direct and tensor product. A characteristic of interaction is that the interaction of two linear or one-dimensional posets is a rectangular or two-dimensional poset.

What we shall verify here is that \otimes is the proper operation for the interaction of those event spaces mimicking posets, namely free event spaces. Given two such event spaces $F(P)$ and $F(Q)$, their the event space mimicking the interaction $P \times Q$ of posets P and Q is $F(P \times Q)$. But this is equal to $F(P) \otimes F(Q)$, showing that at least in the case of free event spaces interaction is \otimes .

There are two programming connectives that do not fit into this system of arithmetic in an obvious way, namely demonic choice and concatenation. Demonic choice of a and b removes ∞_a and ∞_b , takes the disjoint union of the result (as posets), then puts one copy of ∞ back. This description makes it clear that demonic choice is self-dual: $(a \sqcup b)^\perp = a^\perp \sqcup b^\perp$. In Figure 1, $3 = 2 \sqcup 2$, $5 = 3 \sqcup 2$, and $6 = 2 \sqcup 4$. Choice as product is angelic in that the common part of the choice represents decision making; in demonic choice there is no provision for decision making, one takes pot luck. This version of demonic choice, which may not agree with everyone's understanding of the notion (I would appreciate advice on this), is equivalent to the choice having been made "in the beginning," i.e. in the start state, a computational form of original sin.

Concatenation $a; b$ joins a separate copy of b onto each final state of a , by identifying the start state of b with that final state. In Figure 1, $4 = 2; 2$, $7 = 2; 4$, and $9 = 2; 3$. The dual of concatenation is given by $a; b = (a^\perp; b^\perp)^\perp$, its optimal description as far as understanding its meaning goes, and is what concatenation must become for schedules if it is to mean what it does for automata. Unlike demonic choice, concatenation is not self-dual.

We have not investigated the nature of these operations for nonfree event spaces, other than to note that they are well-defined for them. However the nonfree examples we have considered suggest that the interpretation of $a + b$ as concurrence, $a \times b$ as choice, and $a \otimes b$ as interaction continue to make good sense.

7 Event Spaces as a Model of Linear Logic

Linear logic was introduced by Girard in 1987 [Gir87, Gir89]. Its language corresponds exactly to the arithmetic portion of our calculus and thus excludes demonic choice and concatenation. (Though we have been faithful to the basic language of linear logic our choice of notation for its operations and constants is closer to that of Seely [See89] and Barr [Bar91a, Bar91b] than of Girard.)

In assessing the significance of linear logic it is tempting to focus on the closed but not cartesian closed aspect. But that aspect is already present in relation algebras and relevant logic. What is novel and beautiful about linear logic is that, starting from a closed category, in our case event spaces, it adds two useful operations. First an abstraction operator $!$ which forgets enough operations to turn the closed structure into a cartesian closed structure better suited to elementary (“element-oriented”) logic as opposed to categorical logic, in our case posets though we could just as well have chosen sets, and which permits expression of such relationships as $!(a \times b) = !a \otimes !b$ and $b \Rightarrow a = !b \multimap a$. Second an involutory duality a^\perp that gives every monotone operation its De Morgan dual (\otimes , \times , and $!$ are dual to \oplus , $+$, and $?$ respectively) and also makes other useful connections such as $b \multimap a = b^\perp \oplus a$.

We dispense here with the traditional Gentzen sequent formulation of linear logic in favor of a more direct categorical description; consult Seely [See89] for details of the connection between the two.

A model of linear logic is a category C with two closed monoidal structures having $b \multimap a$ and $b \Rightarrow a$ as the respective “internal hom(functor)s” or implications. The left adjoint to $b \multimap -$ is $b \otimes -$, a symmetric operation ($a \otimes b = b \otimes a$), while the left adjoint to $b \Rightarrow -$ is $b \times -$. Furthermore \times is ordinary product in C , which is to say that \Rightarrow confers a cartesian closed structure on C (whence \times is also symmetric). There exists a dualizing object \perp , with the dual a^\perp defined as $a \multimap \perp$ and satisfying $a^{\perp\perp} = a$. The products \otimes and \times have respective De Morgan duals $a \oplus b = (b^\perp \otimes a^\perp)^\perp$ and $a + b = (b^\perp \times a^\perp)^\perp$, the latter being coproduct. The units of \otimes , \times , \oplus , and $+$ are respectively $\top = \perp^\perp$, a final object 1 , \perp (already mentioned), and an initial object 0 .

As Seely points out [See89], the two structures are related by a *comonad* or cotriple $(!, \epsilon, \delta)$ consisting of an endofunctor $! : C \rightarrow C$, and for each object $a \in \text{ob}(C)$ a counit $\epsilon_a : !a \rightarrow a$ and a comultiplication $\delta_a : !a \rightarrow !!a$. The comonad satisfies certain conditions that achieve the effect of an adjunction between C and a category of underlying objects of objects of C [BW85, p.95].

From this we may derive many other properties, for example:

$$\begin{aligned}
 !a \otimes !b &= ((!a \otimes !b) \multimap \perp)^\perp \\
 &= (!b \multimap (!a \multimap \perp))^\perp \\
 &= (b \Rightarrow (a \Rightarrow \perp))^\perp \\
 &= ((a \times b) \Rightarrow \perp)^\perp \\
 &= !(a \times b) \multimap \perp \\
 &= !(a \times b) \\
 a \otimes b &= ((a \otimes b) \multimap \perp)^\perp \\
 &= (b \multimap (a \multimap \perp))^\perp \\
 &= (b \multimap a^\perp)^\perp
 \end{aligned}$$

In the case of event spaces, $C = \mathbf{Ev}$, we have already described the interpretations of all the operations, and it should be clear that the above conditions on a model of linear logic have all been met. It remains only to complete the specification of the comonad, namely $(FU, \epsilon, F\eta U)$ where FU

$= \mathbf{Ev} \xrightarrow{U} \mathbf{Pos} \xrightarrow{F} \mathbf{Ev}$, ε (the family $\langle \varepsilon_a \rangle$ of evaluation maps) is the counit of the adjunction $F \dashv U$, and η (the family $\langle \eta_a \rangle$ of embeddings of generators) is the unit of the same adjunction.

We have argued in outline that the full category \mathbf{Ev} of event spaces, a naturally arising class of structures, and their homomorphisms (in the standard sense) is a model of full linear logic. Moreover product and sum are distinct. We would appreciate hearing about any other equally simple model of linear logic that meets all these conditions.

8 The Birkhoff-Stone Duality of Schedules and Automata

For the benefit of strangers to the Birkhoff-Stone duality of schedules and automata we briefly review its essential features in this section. More detailed tutorials may be found elsewhere including [Pra91].

The simplest case of the duality takes a schedule to be a set X of events, and its dual automaton to be its power set 2^X , forming a Boolean algebra under the operations of union, intersection, and complement relative to X . We take the empty subset of X to be the start state of the automaton and X itself to be the final state. To view 2^X as an automaton in the usual sense, associate to each pair $Y \neq Z$ of states satisfying $Y \subset Z \subseteq X$ a transition corresponding to (i.e. labeled with) the nonempty set $Z - Y$ of events. A path from the initial to the final state via such transitions determines a partition of X into a finite sequence of nonempty blocks, each block representing the parallel execution of the events in that block, with the path as a whole then describing an execution of all the events of X in some order. An alternative view of 2^X as an automaton imposes the additional restriction that transitions be labeled only with singletons, corresponding to a strictly sequential execution of the events of X .

We may usefully enhance such a schedule by equipping it with a partial order \leq . The effect of this enhancement on the dual automaton 2^X is to eliminate certain states, leaving only those states that are subsets Y of X such that if $x \leq y$ in the partial order then $y \in Y$ implies $x \in Y$.

Each of these subsets Y has an obvious representation as its *characteristic function* $f_Y : X \rightarrow 2$ defined as $f_Y(x) = 1$ when $x \in Y$ and 0 otherwise. By taking X^{op} to be the order dual of the partially ordered set (X, \leq) and taking 2 to be ordered via $0 \leq 1$, we see that the condition for Y to be a state is equivalent to the requirement that f_Y be a monotone function or poset map from X^{op} to 2 .

Thus we may take the automaton to be the set $2^{X^{\text{op}}}$ of all monotone functions from X^{op} to 2 . In the special case when X is discretely ordered ($x \leq y$ implies $x = y$, i.e. X is just a set) this automaton is a Boolean algebra, but in general it is a distributive lattice, lacking only the complement operation characteristic of a Boolean algebra.

When X is infinite more can be said. For X discrete, $2^{X^{\text{op}}}$ is a complete atomic Boolean algebra. Complete means that every set \mathcal{Y} of subsets of X has a join or sup, namely its union $\bigcup \mathcal{Y}$, and a meet or inf, namely its intersection $\bigcap \mathcal{Y}$. In particular the empty set of subsets has join \emptyset (bottom) and meet X (top), while the set $2^{X^{\text{op}}}$ of all subsets of X has join X and meet \emptyset . An atom is an element x such that $x \wedge y$ is either x or bottom. Atomic means that every element except bottom is above some atom.

In general $2^{X^{\text{op}}}$ is a *profinite* distributive lattice [Joh82, p.250]. This means that it is complete in the same sense as for Boolean algebras, but the notion of atomic is generalized to the requirement

that every element be the join of a set of compact elements. An element x is *compact* when it is the join only of sets containing x ; equivalently, when the join of the set of elements strictly less than x is itself strictly less than x .

In the case of the distributive lattice $2^{X^{op}}$ the compact elements are exactly those subsets Y of X for which there exists an element $y \in X$ such that no proper subset of Y containing y belongs to $2^{X^{op}}$. In terms of states, these are the states containing an event absent from all earlier states, i.e. the earliest opportunity for that event to occur.

9 Acknowledgments

Rob van Glabbeek turned my attention from the overly abstract n -dimensional complexes of [Pra91] to more concrete and plausible cubical sets, and a phone conversation with Bill Lawvere then further turned it from cubical sets to their dual objects bipointed sets. The connection between that duality and Stone duality then dawned on me, and the additional intuition conveyed by the self-duality of **CSLat** [Lat76, Joh78] led quickly to **Ev**, though the concrete interpretation of the phenomena in **Ev** in terms of unreachable events and initial states took time. That $!a$ even made sense in **Ev** was completely unobvious to me until I had digested the relevant parts of the papers of Barr [Bar91a] and Seely [See89], after which I wondered why I did not see it right away. Thanks also to Michael Barr for much helpful email correspondence concerning monads and comonads. Guo Qiang Zhang, Stefano Kasangian, and Jeremy Gunawardena provided valuable feedback on the paper.

References

- [Bar91a] M. Barr. *-Autonomous categories and linear logic. *Math Structures in Comp. Sci.*, 1(2), 1991.
- [Bar91b] M. Barr. Accessible categories and models of linear logic. *J. Pure and Applied Algebra*, 1991. To appear.
- [Bir33] G. Birkhoff. On the combination of subalgebras. *Proc. Cambridge Phil. Soc.*, 29:441–464, 1933.
- [BW85] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer-Verlag, 1985.
- [CCMP89] R.T Casley, R.F. Crew, J. Meseguer, and V.R. Pratt. Temporal structures. In *Proc. Conf. on Category Theory and Computer Science, LNCS 389*, Manchester, September 1989. Springer-Verlag. Revised version to appear in *Math. Structures in Comp. Sci.*, 1:2, 1991.
- [Dun84] J. M. Dunn. Relevant logic and entailment. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume III, pages 117–224. Reidel, Dordrecht, 1984.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [Gir89] J.-Y. Girard. Towards a geometry of interaction. In *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108, held June 1987, Boulder, Colorado, 1989.

- [Gra81] J. Grabowski. On partial languages. *Fundamenta Informaticae*, IV.2:427–498, 1981.
- [Joh78] P.T. Johnstone. A note on complete semilattices. *Algebra Universalis*, 8:260–261, 1978.
- [Joh82] P.T. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.
- [JT48] B. Jónsson and A. Tarski. Representation problems for relation algebras. *Bull. Amer. Math. Soc.*, 54:80,1192, 1948.
- [Lat76] D. Latch. Arbitrary products are coproducts in complete (V-)semilattices. *Algebra Universalis*, 6:97–98, 1976.
- [NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part I. *Theoretical Computer Science*, 13, 1981.
- [Pra79] V.R. Pratt. Process logic. In *Proc. 6th Ann. ACM Symposium on Principles of Programming Languages*, pages 93–100, San Antonio, January 1979.
- [Pra85] V.R. Pratt. Some constructions for order-theoretic models of concurrency. In *Proc. Conf. on Logics of Programs, LNCS 193*, pages 269–283, Brooklyn, 1985. Springer-Verlag.
- [Pra86] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.
- [Pra91] V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.
- [Pri70] H.A. Priestley. Representation of distributive lattices. *Bull. London Math. Soc.*, 2:186–190, 1970.
- [See89] R.A.G Seely. Linear logic, *-autonomous categories and cofree algebras. In *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 371–382, held June 1987, Boulder, Colorado, 1989.
- [Sto36] M. Stone. The theory of representations for Boolean algebras. *Trans. Amer. Math. Soc.*, 40:37–111, 1936.
- [Sto37] M. Stone. Topological representations of distributive lattices and brouwerian logics. *Časopis Pěst. Math.*, 67:1–25, 1937.
- [Win86] G. Winskel. Event structures: Lecture notes for the advanced course on Petri nets. Technical report, University of Cambridge, July 1986.

Algebraic Specification at Work*

(Abstract)

Egidio Astesiano Alessandro Giovini Gianna Reggio

Dipartimento di Matematica

Università di Genova – Italy

Franco Morando

ELSAG (Elettronica San Giorgio) SpA – Genova

The Problem Concurrent software systems are used in fields where safety and reliability are critical. In spite of these needs the validation of concurrent software tends to rely more on empirical experiments (i.e. testing) rather than on rigorous proofs. This practice, which is already bad for sequential software, becomes awful for concurrent systems. Indeed useful concurrent systems are nondeterministic and the behaviour observed during testing is only the most likely to occur. Therefore we are not sure that a concurrent system will always behave correctly on its test-bed even if, during testing, we have observed a correct behaviour.

We have developed a formal approach with associated tools for handling this situation. This method allows us to express and test exhaustively all possible behaviours of a concurrent program and hence to be sure that a tested concurrent program will always behave correctly on its test-bed. For sake of concreteness, we illustrate its features by means of a particular class of applications and a simple paradigmatic example.

The following is an informal outline of the development phases we have in mind.

Concurrent systems such as an operating system kernel, a real time target architecture or a communication network layer, are programmed using some standard sequential language mixed with specific calls to the underlying system primitives. If we model this mix as a concurrent language \mathcal{L} (as it actually is), then a concurrent system becomes a program p in \mathcal{L} . We need methods for specifying rigorously both the concurrent system and its programming environment in a uniform framework.

During the specification phase a tool set can be used to check specification correctness and consistency. When the specification is ready, programmers or hardware designers, in charge of developing the system, can use a language interpreter to run some test programs and see how the system reacts. Prototyping (i.e. “run and see”) is an informal approach to specification understanding, which is of valuable interest in industrial environments. This enables information technology companies to introduce formal specifications, hiring a limited number of specialists in formal description techniques, while people in charge of implementing specifications can use formal documents (that they probably cannot understand) through the filter of the tool set.

Finally, when the specified system has been implemented, programs running on the concurrent system can be tested using a language interpreter. This application of formal techniques to

*This work has been partially supported by “Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo” of CNR Italy and by the COMPASS ESPRIT-BRA W.G. n. 3264.

concurrent software testing is very interesting since the concurrent language interpreter, if based on a formal specification, can fully exploit the nondeterminism introduced by concurrency.

The Formal Approach It consists of *conceptual tools*, *software tools* and *application development rules*.

Conceptual tools They are essentially a technique for modelling and specifying concurrent systems and languages, based on the *SMoLCS* method and theory (see [AR1,AR2,AR3]). In practical applications we distinguish them in two parts.

Structural Concurrent Design and Specification (SCDS) A structural and hierarchical approach to the design and the formal specification of concurrent systems; by applying *SCDS* we get an abstract concurrent machine, formally expressed by an algebraic specification; we believe that *SCDS* should have the role of structured programming in the concurrent case.

The formal model of processes is that of *labelled transition systems*, with the usual interpretation that a transition $s \xrightarrow{l} s'$ models the fact that a process in the state s can evolve into the state s' performing a transition labelled by l (thus l conveys the only amount of information on the transition which is visible to the external world).

In our approach, labelled transition systems are models of particular algebraic specifications: thus we talk of *algebraic transition systems*. The axioms of the specification of an algebraic transition system are *positive conditional axioms* and are split into two sets: *static* and *dynamic* axioms. Static axioms define the properties of states and labels of the system, while dynamic axioms define the transition relation \longrightarrow .

In general processes are just the basic components of a complex concurrent system, and the *SCDS* approach gives the guidelines for structuring the definition of such a system. The structuring is both *static* (dealing with the structure of the system, built starting from processes and shared static information) and *dynamic* (dealing with the structural definition of the activity of the system, which is derived in three schematic steps from the activities of the process subcomponents).

Formal Specification of Concurrent Languages (FSCL) Given the syntax of a concurrent language, we specify its semantics by some language semantic clauses, which essentially constitute a translation into a language for describing processes, that we call "behaviour language"; then by applying the *SCDS* technique we produce an abstract concurrent machine called "underlying abstract concurrent machine" which gives the semantics of the behaviour language. The semantics of a program written in the concurrent language is thus the semantics of the behaviour obtained via this translation. The underlying abstract concurrent machine represents in some sense the semantics of the "concurrent kernel" of the language.

Software tools Our method is supported by a set of tools, which are used to promote a correct exchange of information between who gives the specification and those that must implement it.

Indeed, first, they help the system specifier to develop a correct and complete specification. while, on the other side, they promote specification transfer between the formal methods specialist and the software engineers. The aim of this approach is to widespread the application of formal specification techniques by introducing some of the benefits of a formal settlement into current programming practice, without requiring dramatic re-education of existing manpower.

The tool set reduces the gap between formal and informal levels since it enables engineers to inspect the behaviour of the system they have to implement without bumping into formal

details. Indeed the tools enable the *formal execution* of programs written in a concurrent language \mathcal{L} .

This is done by two tools: the *Concurrent Language Translator* (CLT) and the *Concurrent Rapid Prototyping* (CRP). First the translator CLT takes an \mathcal{L} -program p and uses the language semantic clauses for \mathcal{L} to translate p into bh (a state of the underlying abstract concurrent machine). The prototyper CRP takes then the state bh and using the specification of the underlying abstract concurrent machine generates a tree containing all possible execution paths of bh ("execution tree").

CRP allows an engineer to have a global view of the system behaviour: he/she can examine the whole tree, can check for deadlocks or termination and so on. A global view of the system helps to locate problems. When the user finds an unexpected behaviour, he can switch to a more accurate analysis and walk up and down the execution tree inspecting every node. In this way both programmers and specifiers can understand more deeply what the system can do in a given state, how it can get there and (possibly) what is wrong with it.

This interaction scheme implies an offline generation of part of the execution tree before starting the interactive session. For this reason CRP actually includes two distinct utilities: the "tree builder" and the "tree walker". The two utilities are compatible: the user can traverse the tree while it is being built and he can decide to stop the tree builder when he has obtained relevant information.

The tree builder is a non-interactive system, so that the generation of the tree can proceed offline without any assistance. This feature is especially desirable since tree generation for non-trivial applications can take some amount of time.

The tree walker, instead, is an interactive system that reads the file produced by the tree builder and shows to the user the part of the tree developed so far. In the tree walker one can concentrate all the "bells and whistles" of a friendly user interface.

Application development rules They are application dependent and consist of rigorous guidelines using the conceptual and software tools for the correct development of a class of applications. In the full paper [AGMR] we illustrate the rules for a particular class, by means of a very simple example (but note that the technique is currently used in real industrial practice).

We show how our approach can be used to develop a correct implementation of a semaphore using a shared bit. We have a concurrent schematic language $\mathcal{EL}^{[x]}$ handling access to a shared resource parameterized over the mechanism x for regulating the access. Then we consider two actualizations of $\mathcal{EL}^{[x]}$: the language $\mathcal{EL}^{[Sem]}$, where the mechanism x is expressed by means of a semaphore Sem with the usual primitives P and V, and the language $\mathcal{EL}^{[Bit]}$, where the mechanism x is now a shared bit, with primitives read, write, and test&set.

Now we want to find the $\mathcal{EL}^{[Bit]}$ code corresponding to P and V, i.e. to find $Pbit$ and $Vbit$ fragments of $\mathcal{EL}^{[Bit]}$ programs s.t. for all p in $\mathcal{EL}^{[Sem]}$ we have that p and $p[Pbit/P, Vbit/V]$ could reasonably be considered "equivalent".

To solve this problem we proceed throughout the following phases.

Formal specification of $\mathcal{EL}^{[x]}$. We give the parameterized underlying abstract concurrent machine $UACM[x]$, describing the concurrent structure of $\mathcal{EL}^{[x]}$, and the parameterized language semantic clauses $LSC[x]$, translating $\mathcal{EL}^{[x]}$ programs into states of $UACM[x]$.

Formal specification of $\mathcal{EL}^{[Sem]}$ and $\mathcal{EL}^{[Bit]}$. We give the formal definitions of the two languages by instantiating $UACM[x]$ and $LSC[x]$ respectively with a semaphore specification and with a shared bit specification.

Specification checking and understanding by means of tools. Using the tools we can check whether the specifications given above are adequate; moreover by experimenting with them we can get a better understanding of the object to be implemented and of the implementation language.

Trial implementations. In the full paper [AGMR] we propose some implementations of P and V and check whether they are adequate or not; we first propose an apparently reasonable implementation and by using the tools show that it is not correct and after we look and find a better one.

References

- [AR1] Astesiano E.; Reggio G. "SMoLCS Driven-Concurrent Calculi", *Proc. TAPSOFT'87. Vol. I*, Lecture Notes in Computer Science n. 249, Springer Verlag, 1987.
- [AR2] Astesiano E.; Reggio G. "Direct Semantics for Concurrent Languages in the SMoLCS Approach", *IBM Journal of Research and Development*, 31, 5 (1987), pp. 512-534.
- [AR3] Astesiano E.; Reggio G. *A Structural Approach to the Formal Modelization and Specification of Concurrent Systems*, Technical Report n. 0, Formal Methods Group, University of Genova, 1990.
- [AGMR] Astesiano E.; Giovini A.; Morando F.; Reggio G. *Algebraic Specification at Work*, Technical Report n. 11, Formal Methods Group, University of Genova, 1990.

ON THE REUSABILITY OF SPECIFICATIONS AND IMPLEMENTATIONS

Francesco Parisi-Presicce

Department of Mathematics
University of Southern California
Los Angeles, CA 90089-1113

on sabbatical leave from Dipartimento di Matematica Pura ed Applicata
Universita' degli Studi L'Aquila
I-67100 L'Aquila Italy

The problem of software reusability is not new, but it has become even more important to reduce the development cost and improve the reliability of software in increasingly more complex systems. A formal specification is crucial to identify and correctly employ the reusable components and the algebraic framework for such specifications provides useful methods for program verification, structuring mechanisms for system configuration and the possibility to deal with different levels of abstraction.

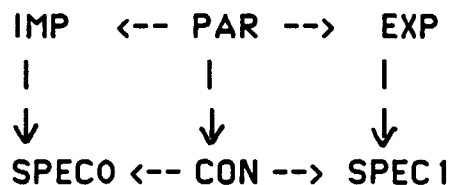
By representing the visible part of a module specification [EW85] by a production of specifications [PP90], we have addressed the problem of reusing module specifications and in particular the problem of designing a modular system, with given interfaces, by using only the module specifications of a given library. In the original formulation [PP90], the specifications are algebraic, of the form (S, OP, E) with (S, OP) a signature and E a set of (positive conditional) equations.

A module specification consists of visible import and export interfaces with a shared parameter part $IMP \leftarrow PAR \rightarrow EXP$ and a body BOD , which specifies an implementation of the export interface EXP by the import interface IMP and is related to the interfaces as in the commutative diagram

$$\begin{array}{ccc} PAR & \dashrightarrow & EXP \\ \downarrow & & \downarrow \\ IMP & \dashrightarrow & BOD \end{array}$$

Given a library LIB of module specifications, it is easy to define a set $PROD$ of algebraic specification productions of the form $IMP \leftarrow PAR \rightarrow EXP$ by taking the visible part of each module in LIB . The notion of derivability $SPECO \Rightarrow SPEC1$ is then formally defined by the double pushout diagram

in the category of algebraic specifications



Then the problem of designing a modular system SYS with overall interfaces IMP^* and EXP^* , which reuses only the modules in LIB is equivalent to the derivability of EXP^* from IMP^* using the productions in PROD in the sense of [EHKP91]. Furthermore, the formal derivation provides the actual interconnections of the modules needed in the design of the system SYS . Notice that the notion of derivation is symmetric with respect to IMP and EXP and therefore the derivation process can be goal-directed (given the objective EXP , derive a specification IMP which is immediately realizable) or generative (given a realized specification IMP , generate all possible specifications which are realizable using the given modules).

The first immediate 'reuse' of this approach is in the problem of implementing a specification by another one. The body part BOD plays no role in the derivation of a specification by another one, but only in the design of the final modular system. We can think of a production $\text{SP} \leftarrow \text{P} \rightarrow \text{SP}'$ as representing the fact that SP' is an implementation of SP . In [WHB88], for example, $\text{SP} \rightsquigarrow \text{SP}'$ (SP is implemented by SP') if $\text{S} \subset \text{S}'$, $\text{OP} \subset \text{OP}'$ and $\text{A} \models_{\text{SP}}$ is a model of SP for all models A of SP' . We can immediately translate this relation into a production by taking $\text{P} = (\text{S}, \text{OP}, \{\})$. Then the implementability of SPEC0 by SPEC1 reusing preexisting implementations is equivalent to the derivability of SPEC1 from SPEC0 . In order for this approach to be useful, there ought to be a way to automatically translate the derivation sequence into the derived implementation. In the case above, the translation is immediate. In [WHS89], reusable components are represented by trees, where a specification SP' is a child of another one SP if SP' is an implementation of SP . Each tree is easily translated into productions, by associating a production $\text{SP} \leftarrow \text{P} \rightarrow \text{SP}'$ to each edge of the tree. The transitivity of the implementation relation is then reflected by the composability of the productions obtained in this translation as follows. If $\text{SP} \rightsquigarrow \text{SP}'$ then $\text{S} \subset \text{S}'$ and $\text{OP} \subset \text{OP}'$ and $\text{SP} \leftarrow \text{P} \rightarrow \text{SP}'$ as above; if $\text{SP}' \rightsquigarrow \text{SP}''$, then $\text{S}' \subset \text{S}''$ and $\text{OP}' \subset \text{OP}''$ and $\text{SP}' \leftarrow \text{P}' \rightarrow \text{SP}''$. But then $\text{SP} \leftarrow \text{P} \rightarrow \text{SP}''$ is the production representing $\text{SP} \rightsquigarrow \text{SP}''$.

One advantage of this production approach is that it does not need a structured specification to start with. All specifications are intended to be "flat" as a way to represent them in normal form. In [WHS89], the abstract specification SP is assumed to be decomposed into subspecifications SP_1, \dots, SP_n such that $SP = t [SP_1, \dots, SP_n]$ where t is a term constructed with specification-building operations; then the SP_i are matched against the reusable components SP_i' and then the implementation SP' of SP is reconstructed by $t [SP_1', \dots, SP_n']$. This approach may require that more than one decomposition of SP be tried, and still there is no guarantee that one can be found that allows reusability of given components. For example, SP may be decomposed into $t [SP_1, \dots, SP_n]$ but none of the components SP_i may match an already known implementation SP_i' and the problem remains even if properties of the specification operations are used to restructure the specification SP into $t' [SP_1, \dots, SP_n]$. It may be possible to match known implementations only if subspecifications of SP are allowed to be "flatten", increasing considerably the number of decompositions of SP to be tried. In the production approach, the derivation itself provides the term t and its arguments SP_1', \dots, SP_n' . In this case, of course, the 'applicability' of a production to a specification is more expensive because the whole specification SP must be checked to find the occurrence morphism $SP_i \rightarrow SP$ for the application of $SP_i \leftarrow P_i \rightarrow SP_i'$ to SP (provided the 'Gluing Conditions' are satisfied [EP91]).

We do not advocate always the use of flat specifications. In fact, structured specifications can be exploited to carry out proofs of validity of formulas [W90]. This brings us to another reuse of the approach: parsing of an algebraic specification. Just as in classical languages, the productions of a specification grammar [EP91] can be used to decompose flat specifications. Once the specification is structured, the flat proof system can be structured too.

Both the parsing and the reuse of implementations would benefit from the flexibility provided by transformations of specifications from one institution to another one. In particular in the process of implementing specifications, a sequence of steps $SP \rightsquigarrow SP'$ is intended to provide more "concrete" specifications of the abstract one, until one in "executable" form is found. This corresponds to having a specification SP in an institution INS and wanting an implementation SP' in an institution INS' .

An institution INS consists of

- a category Sig_{INS} of signatures
- a functor $Sen_{INS} : Sig_{INS} \rightarrow Set$ giving the set of sentences over a

given signature

- a functor $\text{Mod}_{\text{INS}} : \text{Sig}_{\text{INS}} \rightarrow \text{Cat}^{\text{OP}}$ giving the category of models of a signature
- a satisfaction relation $\models \subset |\text{Mod}_{\text{INS}}(\Sigma)| \times \text{Sen}_{\text{INS}}(\Sigma)$

Among examples of institutions are equational logic, first order logic and Horn clause logic.

Having already verified that a specification SPi written in an institution INS can be implemented correctly by a specification SPi' in another institution INS' , we can formalize this relation for future "reuse" by first identifying a common subspecification SPi^* defined in a "subinstitution" INS^* of both INS and INS' , and then represent the implementation as a pair of specification morphisms $\text{SPi}^* \rightarrow \text{SPi}$ (in INS) and $\text{SPi}^* \rightarrow \text{SPi}'$ (in INS').

Having a library $\{ \text{SPi} \leftarrow \text{SPi}^* \rightarrow \text{SPi}' \}_1$ of previously verified correct implementations, we can view the problem of implementing a given specification SP in INS into some SP' in INS' , using only the verified implementations, into the problem of deriving from SP a specification in INS' using $\text{SPi} \leftarrow \text{SPi}^* \rightarrow \text{SPi}'$ as productions. If SP can be transformed into a specification in INS' , then the derivation provides the appropriate combination of the SPi' (based on given specification-building operations) to obtain the implementation SP' .

The derivation must take place in an institution which allows both the specifications SPi and SPi' as follows.

The "subinstitution" INS^* is determined by two institution morphisms $\Phi : \text{INS} \rightarrow \text{INS}^*$ and $\Phi' : \text{INS}' \rightarrow \text{INS}^*$

where an institution morphism $\Phi : \text{INS} \rightarrow \text{INS}^*$ consists of

- a functor $\phi : \text{Sig} \rightarrow \text{Sig}^*$
- a family of functions $\alpha_{\Sigma} : \text{Sen}^*(\phi(\Sigma)) \rightarrow \text{Sen}(\Sigma)$ [a natural transformation]
- a family of functors $\beta_{\Sigma} : \text{Mod}(\Sigma) \rightarrow \text{Mod}^*(\phi(\Sigma))$ [a natural transformation] compatible with the satisfaction relations \models and \models^* ([GB83])

As the institution morphisms map richer institutions into more primitive ones, an institution INS^* that combines INS and INS' can be defined by a limit construction ([T86]). In such an institution, properties, say of operator symbols, can be expressed as sentences of either INS or INS' .

The derivation $\text{SP} \Rightarrow \text{SP}'$ from INS to INS' takes place in this limit institution INS^* .

ACKNOWLEDGMENTS Part of this work was conducted at the Facultat fur Mathematik und Informatik of the Univ. Passau and at the Dept. of Elect. Engin. of the Univ. of Southern California. It was partially supported by CNR under "Progetto Finalizzato : Sistemi informatici e Calcolo Parallelo"

REFERENCES

- [EHKP91] Ehrig H., Habel A., Kreowski H.-J., Parisi-Presicce F., From Graph Grammars to High Level Replacement Systems, Proc.4th Intern.Workshop on Graph Grammars, to appear as Lect.Notes Comp.Sc.
- [EM85] Ehrig H., Mahr B., Fundamentals of Algebraic Specifications 1, Springer-Verlag, 1985
- [EP91] Ehrig H., Parisi-Presicce F., Algebraic Specification Grammars, Proc. 4th Intern.Workshop on Graph Grammars, to appear as Lect.Notes Comp.Sc.
- [EW85] Ehrig H., Weber H., Algebraic Specification of Modules, in 'Formal Models in Programming' (eds. Neuhold E.J., Chronist G.) North-Holland 1985
- [GB83] Goguen J.A., Burstall R.M., Introducing Institutions, Proc. Logics of Programs 1983, Lect.Notes Comp.Sc. 164 (1983) 221-256
- [PP90] Parisi-Presicce F., A Rule-Based Approach to Modular System Design, Proc 12th Int.Conf.Soft.Eng. (1990) 202-211
- [ST85] Sannella D., Tarlecki A., Specifications in arbitrary institutions, CSR-184-85, U.Edinburgh
- [T86] Tarlecki A., Bits and pieces of the theory of institutions, Proc. Wksp Category Theory and Comp. Progr. 1985, Lect.Notes Comp.Sc. 240 (1986) 334-363
- [WHS89] Wirsing M., Hennicker R., Stabl R., MENU - An example for the systematic reuse of Specifications, ESEC 89, Lect.Notes Comp.Sc. 387
- [WHB88] Wirsing M., Hennicker R., Breu R., Reusable specification Components, MFCS 88, Lect.Notes Comp.Sc. 324 (1988) 121-137
- [W90] Wirsing M., Proofs in Structured Specifications, Univ. Passau Techn. Report 1990

Selecting Reusable Components Using Algebraic Specifications

Extended Abstract

David Eichmann

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
email: eichmann@a.cs.wvu.wvnet.edu

1. Introduction

A significant hurdle confronts the software reuser attempting to select candidate components from a software repository – discriminating between those components without resorting to inspection of the implementation(s). We outline an axiomatic approach to this problem based upon Gutttag and Horning's algebraic specification techniques [3]. This approach selects candidates using both their interfaces, using signatures, and their behavior, using axioms.

2. The Problem

A mature software repository can contain thousands of components, each with its own specification, interface, and typically, its own *vocabulary*. Consider the signatures presented in figures 1 and 2 for a stack of integers and a queue of integers, respectively. These signatures are isomorphic up to renaming, and thus exemplify the *vocabulary problem*. Software reusers implicitly associate distinct semantics with particular names, for example, pop and enqueue. Thus, by the choice of names, a component developer can mislead reusers as to the semantics of components or provide no means of discriminating between components. Renaming push enqueue, pop dequeue, and top front in a stack component is an example of the former. Renaming push and enqueue insert, pop and dequeue remove, and top and front current in a stack component and a queue component, respectively, is an example of the latter.

3. Related Work

Recent proposals for repository interfaces [2,4,6] fail to address the vocabulary problem, since they concentrate on vocabulary-oriented classification techniques, e.g., from library science. Prieto-Diaz and

```
Create:  → Stack
Push:    Stack × Integer → Stack
Pop:     Stack → Stack
Top:     Stack → Integer
Empty:   Stack → Boolean
```

Figure 1 – Signature for the Stack Specification

```
Create:  → Queue
Enqueue: Queue × Integer → Queue
Dequeue: Queue → Queue
Front:   Queue → Integer
Empty:   Queue → Boolean
```

Figure 2 – Signature for the Queue Specification

```

Pop(Push(S, I)) = S
Top(Push(S, I)) = I
Empty(S) = if (S == Create) then true else false

```

Figure 3 – Axioms for the Stack Specification

```

Dequeue(Enqueue(Q, I)) = if (Q == Create) then Create
                        else Enqueue(Dequeue(Q), I)
Front(Enqueue(Q, I)) = if (Q == Create) then I
                      else Front(Q)
Empty(Q) = if (Q == Create) then true else false

```

Figure 4 – Axioms for the Queue Specification

Freeman used the notion of literary warrant to develop the faceted classification approach [6]. They clustered descriptive terms drawn from sample components in the repository into a number of facets comprising a single tuple schema. The faceted classification approach suffers from the vocabulary problem due to the probable ambiguity in the vocabulary used both in the components and the corresponding documentation. For example, consider the case where various components use terms such as destroy, delete, remove, discard, etc. – all pairwise synonyms, but with quite distinct semantics.

Eichmann and Atkins further structured the facets and facet tuples into a lattice, alleviating the requirement that all components contain a value for all facets [2]. The classification of a component contained a set of values drawn from a given facet, avoiding the need to compute closeness metrics.

Neither of the above approaches completely overcomes the true nature of the vocabulary problem, the issue of behavior. Algebraic specification techniques (e.g., [3]) partially (and unintentionally) overcome the vocabulary problem through inclusion of behavioral axioms into the specification. Figures 3 and 4 provide characterizations for figures 1 and 2, respectively (ignoring error semantics for the sake of simplicity). The main objection to algebraic specifications is in the need to *comprehend* the specifications retrieved from the repository. The traditional examples in the literature rarely exceed the complexity exhibited in figures 1 and 2.

4. Behavior Specifications in Reuse

We propose a repository retrieval interface based upon both the vocabulary used in components and the *observable behavior* of components, that is, the axioms that formally characterize the semantics of components. The ability for a reuser to partially specify component behavior is a key element of the interface design.

The repository itself consists of the actual components (we aren't concerned at this point whether they are stored in source form, or in executable form (as considered by Weide, et. al. [7])), axiomatic specifications for each of the components, and a vocabulary-based classification structure.

Retrieval of components under this system proceeds in two phases. A reuser initially specifies a vocabulary-based query, narrowing the field of candidates to those that are isomorphic to the query signature. The axioms characterizing each of the candidate components in turn are then used as theories supporting attempted proofs of the proposition(s) the reuser poses in the second phase of the query (using an existing theorem prover, e.g., RRL [5]). Successful proof of all of the propositions posed by the user indicates that the component of interest provides at least the semantics sought after.

This by no means implies that the components thus retrieved have the same semantics. For example, the query proposition

```

Create:  → Stack
Push:    Stack × Integer → Stack
Pop:     Stack → Stack
Top:     Stack → Integer
Empty:   Stack → Boolean
Depth:   Stack → Integer

```

Figure 5 – Signature for the Stack Specification

$\text{Remove}(\text{Insert}(\text{Create}, x)) = \text{Create}$

can be proven both by the stack axioms (with Remove bound to Pop and Insert bound to Push) and by the queue axioms (with Remove bound to Dequeue and Insert bound to Enqueue). However, the query proposition

$\text{Remove}(\text{Insert}(\text{Insert}(\text{Create}, x), y)) = \text{Insert}(\text{Create}, x)$

can be proven only by the stack axioms; having failed to prove the query proposition, the queue specification would be removed as a candidate. Our assumption in this approach is that the reuser will pose propositions that best characterize the behavior of interest (i.e., the second example proposition better characterizes a stack than does the first example proposition), thereby providing better discrimination between signature-isomorphic components.

Propositions posed by reusers need to be tested against a single specification's axiom set multiple times in cases where an operation from the reuser's query signature cannot be resolved to a single operation in a candidate component's signature. This usually results from an insufficient vocabulary framework. Consider the signature of figure 5, a slightly extended version of figure 1. In the absence of any classification information specifically concerning the Top and Depth operations for query propositions such as

$\text{Query}(\text{Insert}(\text{Insert}(\text{Create}, x), y)) = y$

$\text{Query}(\text{Insert}(\text{Insert}(\text{Create}, x), y)) = 2$

(assuming that these two propositions are posed in separate queries) the system must attempt a proof of the proposition using both a binding of Query to Top, Insert to Push, and Create to Create, and a binding of Query to Depth, Insert to Push, and Create to Create. The first proposition is successfully proved using the first binding and the second proposition by the second binding.

5. Conclusions

Our approach merges traditional vocabulary and syntactic based retrieval mechanisms with the formal semantics of algebraic specification. Neither retrieval mechanism in isolation is sufficient to completely address the entire problem. The complete paper discusses in more detail the nature of the retrieval mechanism, in particular, the retrieval of candidate components that are isomorphic to some subtype of the query signature [1].

References

- [1] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pages 471–522, December, 1985.
- [2] D. Eichmann and J. Atkins, "Design of a Lattice-Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, pages 90–97, June 21–23, 1990.
- [3] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pages 27–52, 1978.
- [4] W. P. Jones, "On the Applied use of Human Memory Models: The Memory Extender Personal Filing System," *Int. Journal of Man-Machine Studies*, vol. 25, no. 2, pages 191–228, August, 1986.

- [5] D. Kapur and H. Zhang, "RRL: A Rewrite Rule Laboratory," *Ninth International Conference on Automated Deduction (CADE-9)*, Argonne, IL, May, 1988.
- [6] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pages 6-16, 1987.
- [7] B. Weide, W. Ogden, S. Zweben, "Reusable Software Components," *Advances in Computers*, M. C. Yovits, ed., Academic Press, 1991.

The combination of specifications and the induced relations in object oriented programs

G Steve Hirst and T B Dinesh
Department of Computer Science,
University of Iowa.
{hirst,dinesh}@cs.uiowa.edu

1 Introduction

This paper investigates a declarative method for the specification of objects in an object oriented language. This specification method constructs specifications by combining sub-specifications and elaborating specification modules. Using this specification system as an example we examine the more general concepts of inheritance and re-use in order to formally model these processes. This paper is the product of two separate research agenda. One involves investigating the semantics of logic programming languages in order relate the pragmatics of logic programming to the static structure of logic programs. This has lead to a model of logic programs based on the combination of substitutions. The other involves studying the behavior of object oriented programs to allow for the automatic generation of coercions for operations between objects which are subtypes of a common type. This paper is an outgrowth of some of the common themes that appear to arise in both these research programs.

This paper presents our view of object as a "state space" which constrains its behavior. The language "Suspect" is used as a simple specification language for objects; this language contains operators for combining specifications and for abstracting over specifications. The paper examines the relations induced in objects by their composition from other objects and how these correspond to concepts of class and inheritance. The paper also begins a discussion of the algebraic foundations of this view of specification.

2 Specifying objects: 'Objects as State Space'

Objects fill the role of values in object-oriented languages; they can be assigned as values, passed as parameters, and are *typed*. They are more complicated than the values generated by an abstract data type in that they posses a *state* which alters as a result of operations on them. Just as we can specify a data type without providing representational details for the values; we would like to be able to specify an object abstractly without providing the procedural details of instance variables and their relationship to the methods applied to the object.

Instead of the object *containing* state variables; we will specify objects as constraints on their *state space*. For example, a **rectangle** might have as its attributes the coordinates of its four corners. There may be methods which modify one or more of these attributes but certain constraints should continue to hold between the corners. We might require that the rectangle be defined by two opposing corners. Therefore the horizontal coordinate of the top left corner must be the same as the horizontal coordinate of the top right. This constant relationship between the attributes of an object can be defined by a set of constraints on the attributes. For this example the constraints are simple equality constraints on components of each attribute.

In this view, an object is defined only by the state space it occupies. The details of the methods which manipulate this state or project the state of one object into another object are not directly accessible. Some of the methods which modify the internal state of an object can be seen as a matter of additional constraints on the object's state space. An example of such a method is expansion of a rectangle to a smallest enclosing square. Other methods must be specified as objects which relate the two states of the object (before the method application and after the method application); a **MoveDown** method could therefore be specified

as an object which constrains the values of the corners after the move in terms of the corners before the move

3 A specification language: "Suspect"

We use a specification language "Suspect" based on equation solving of Herbrand terms. We use the convention that variable symbols in the Herbrand terms begin with uppercase letters; constant and function symbols begin with lowercase letters. A "suspect" specification for the object **rectangle** discussed above is given as:

$$\left\{ \begin{array}{l} P1 = pt(X0, Y0) \\ P2 = pt(X1, Y0) \\ P3 = pt(X1, Y1) \\ P4 = pt(X0, Y1) \end{array} \right\}$$

The solutions of an equational specification are *grounding substitutions* over the Herbrand universe. [LMM88]. A grounding substitution is a map from a set of variable symbols to a set of variable free Herbrand terms. For example, the following grounding substitution is a solution to the rectangle specification in the sense that applying the substitution to the variable symbols in the equation set makes each equation a syntactic identity.

$$\{P1 \rightarrow pt(1, 1), P2 \rightarrow pt(4, 1), P3 \rightarrow pt(4, 4), P4 \rightarrow pt(1, 4), X0 \rightarrow 1, X1 \rightarrow 4, Y0 \rightarrow 1, Y1 \rightarrow 4\}$$

The *solution set* of a specification is the set of all grounding substitutions which make the lefthand and righthand side of each equation syntactically identical. "Suspect" allows specifications to be built by combining operators applied to "sub-specifications." The most basic such operation is the joining of equation sets. For example, the **rectangle** object after its top left corner was fixed (perhaps by a method application) could be specified by combining the equation set given above with the additional equation $\{P1 = pt(1, 1)\}$. The solutions to this combined set of equations is exactly equal to the intersection of the solution sets of each of the components.

Combining solution sets by taking their intersection allows us to construct more constrained objects from less constrained objects. This is clear in the above example where a rectangle object is "specialized" by fixing one of its corners. Constructing complex behaviors from simple ones requires a combining operation which builds less constrained objects. The *alternation* operator ($()$) in "suspect" performs this function. For example, we may wish to constrain the values of the coordinates of a point to some set of values such as a range of integers. The equation set $\{X = 0\}$ has as its solutions all grounding substitutions which map X to 0. Another equation set $\{X = 1\}$ has the set of all grounding substitutions which map X to 1 as its solutions. Combining these two specifications with the alternation operator produces a new specification whose solutions are the union of the solution sets of the original specifications. A sequence of alternations constrains the variable X to one of a set of values.

The notion of combining sets of equations can be extended to combining specifications built with the alternation operator by defining a *joining* operator ($+$) as the intersection of the solutions to the combined specifications. Now we can specify a Point with coordinates X and Y with the following expression:

$$(\{X = 0\}|\{X = 1\}|\dots|\{X = c\}) + (\{Y = 0\}|\{Y = 1\}|\dots|\{Y = c\})$$

This specification has as its solution set all grounding substitutions which bind both X and Y to a constant from the set $\{0, 1, \dots, c\}$.

4 The interpretation of objects and their induced relations

In the example above, we specified a point object whose attributes X and Y were constrained to take their values from some range of coordinate values. The component of the specification dealing with X and Y differ only by the particular variable symbol involved. The constraints on both variables are *similar* in this regard. This relationship is made explicit by the notion of specification *module*. A module is just an abstraction of a specification with the actual variables left of $::$ as parameters. The module *COORD* is an abstraction of the constraints on both X and Y :

$$COORD < V > :: \{V = 0\} \{V = 1\} \dots \{V = c\}$$

And the individual specifications are written as *elaboration* expressions such as $COORD < X >$ and $COORD < Y >$ which represent a textual substitution of the arguments for the corresponding variable in the module specification. These elaboration expressions can be used with combination operators to build more complex specifications and modules. Hence, the specification of the point with coordinates X and Y is $COORD < X > + COORD < Y >$. Furthermore, the module for all such point objects can be specified as:

$$POINT < Xcoord, Ycoord > :: COORD < Xcoord > + COORD < Ycoord >$$

The concept of method specialization is both important and problematic to the designers of object oriented languages. There is a parallel between the use of specification combinations to reuse and refine object specifications and the notion of method specialization and thus inheritance in object oriented languages. The phenomena of method specialization has been studied by looking at natural transformations in a preorder category [Mit90].

An object representing a point on a display is frequently used to explain inheritance and method specialization [CM89]. In this system, the specification of **POINT** discussed above can be combined with the specification of motion in a coordinate system and the result is the specification of a movable point. One way in which to specify a movable point is as an object which relates a pair of points. Given a specification of the relation **INCR** between two coordinates $X0$ and $X1$ as $INCR < X0, X1 >$, the specification can be written as:

$$INCR < Y0, Y1 > + \{X0 = X1\} + POINT < X1, Y1 >$$

This expression specifies that the pair $X0, Y0$ is constrained by the values of a point attributes $X1, Y1$ such that $Y0$ is greater than $Y1$ by one and $X0$ equals $X1$. If we are interested in the specification of a moved point, it is useful to *project* away the information about the variables $X1$ and $Y1$, i.e., to dispose of any constraints on the bindings of the variables. We can do this by constructing a module from this specification which does not provide a parameter value for $X1$ and $Y1$.

$$MOVE < X, Y > :: INCR < Y, Y1 > + \{X = X1\} + POINT < X1, Y1 >$$

Elaborating the specification $MOVE < X0, Y0 >$ creates new variables for $X1$ and $Y1$ whose binding can safely be ignored in the specification of constraints on $X0$ and $Y0$.

A sub-class of point can be specified by combining the point specification with a set of constraints on more variables as in:

$$CPOINT < X, Y, C > :: POINT < X, Y > + (\{C = red\} \{C = green\} \{C = blue\})$$

Taking the join of two specifications produces a specification which is at least as constrained as either of the components. In this example any solution (grounding substitutions) for **CPOINT** is also a solution to **POINT**. Specifying the constraints on a moveable color point is now just a matter of an expression combining the constraints on any moveable point with those of a colored point:

$$CMOVE < X, Y, C > :: CPOINT < X, Y, C > + MOVE < X, Y >$$

The specifications elaborated from this module, are specialized versions of specifications elaborated from **MOVE** in that any ground substitutions which solves the colored move specification will also solve the original move specification. This is also a case of reuse in that the behavior defined by **MOVE** is used unchanged by colored move; if the definition of **MOVE** were to change (to include other directions of movement for example) that resulting change in constraints would automatically be reflected in the specifications based on **CMOVE**.

5 Algebraic foundations

Let S be the substitution system with a set $|S|$ of substitutions and a set $||S||$ of types, a partial composition operation on $|S|$ denoted by $;$ and the *source* and *target* operations denoted $\neg, \neg : |S| \rightarrow ||S||$, is defined as in [Gog88]. However, to express the idea that constraints get stronger over "method applications" we use

indexed variables, with the interpretation that variables with higher indices are at least as constrained as the lower ones.

If $\{x_1, x_2, \dots\}$ and $\{y_1, y_2, \dots\}$ are two indexed sets of variables, then a substitution system $S_{x,y}$ consists of objects that are sets of variables indexed by natural numbers with morphisms from $S_j \rightarrow S_i$ where S_i does not contain any x_j or y_j with $j > i$. In particular, the identity morphism has $i = j$.

Let ANY be the set of finite variables, Z be a subset of ANY and S_{ANY} be an arbitrary substitution system (that indexes its variables). Then " Z " specifications are functors from $S_{ANY \setminus Z}$ to S_{ANY} .

For example, we say that *Point*, as defined by the module *POINT*, is a functor $S_{c,d,r} \rightarrow S_{x,y,c,d,r}$ for some $\{x, y, c, d, r\}$. We call the morphisms in $S_{x,y,c,d,r}$ simply *points* and the morphisms in $S_{c,d,r}$ *non-points*. A specification of **COLOR**

$$\{C = red\} \mid \{C = green\} \mid \{C = blue\}$$

corresponds to a *non-point* morphism $N_i \rightarrow N_j$ for $N_i, N_j \in ||S_{c,d,r}||$ and to a *point* morphism $P_k \rightarrow P_l$ for $P_k, P_l \in ||S_{x,y,c,d,r}||$ and $i \geq j$ and $k \geq l$.

Following [Mit90], "moving" points is a natural transformation, when we consider

$$INCR < Y_{i+1}, Y_i > + \{X_{i+1} = X_i\}$$

as $move : Z_{i+1} \rightarrow Z_i$ to be

$$\{x_{i+1} \rightarrow x_i, y_{i+1} \rightarrow y_i + 1\} \cup \{z_{i+1} \rightarrow z_i \mid z_{i+1} \in Z_{i+1} \setminus \{x_{i+1}, y_{i+1}\}\} \cup \{z_k \rightarrow z_k \mid z_k \in Z_{i+1}, k < i + 1\}$$

POINT and *MOVE* defined above correspond to two functors from *non-points* to *points* with *move* being a natural transformation.

CPOINT corresponds to a functor $S_{d,r} \rightarrow S_{x,y,c,d,r}$, from *non-color-points* to *color-points*, while preserving the *POINT* morphisms in $S_{x,y,c,d,r}$. Thus *color-points* are a full subcategory [Gog88] of *points*, with the "method" *move* defined on *points* specialized to *color-points*.

Acknowledgements: We thank Art Fleck for his assistance in the revision of this paper.

References

- [CM89] Luca Cardelli and John Mitchell. Operations on records. In *Proceedings of the Fifth Conference on Mathematical Foundations of Programming Language Semantics*, 1989.
- [Gog88] Joseph A. Goguen. What is unification? a categorical view of substitution, equation and solution. Technical Report CSLI-88-124, CSLI, CSLI/Stanford Ventura Hall Stanford, CA 94305, April 1988.
- [LMM88] J.-L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.
- [Mit90] John Mitchell. Towards a typed foundation for method specialization and inheritance. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages*, 1990.

MEC : a system for constructing and analysing transition systems

André Arnold
Laboratoire d'Informatique *
Université Bordeaux I

Abstract

MEC is a tool for constructing and analysing transition systems modeling processes and systems of communicating processes.

From representations of processes by transition systems and from a representation of the interactions between the processes of a system by the set of all allowed global actions, MEC builds a transition system representing the global system of processes as the *synchronized product* of the component processes.

Such transition systems can be checked by computing sets of states and sets of transitions verifying properties given by the user of MEC. These properties are expressed in a language allowing definitions of new logical operators as least fixed points of systems of equations; thus all properties expressed in most of the branching-time temporal logic can be expressed in this language too.

MEC can handle transition systems with some hundred thousands states and transitions. Constructions of transition systems by synchronized products and computations of sets of states and transitions are performed in time linear with respect to the size of the transition system.

Introduction

The notion of *transition system* plays an important role for describing and studying processes and systems of communicating processes. A simple way to represent processes, introduced for instance in [10] and widely used in many works on semantics and verification of processes, is to consider that a process is a set of *states* and that an *action* or an *event* makes the current state of the process to change; thus the possible elementary behaviours of the process are represented by *transitions*: each transition contains the current state of the process, the new state it enters and the name of the action or event which caused this change. Transition systems are also used to describe systems of communicating processes, and not only individual processes; the states of the system are the tuples of states of its components and the transitions of the systems are tuples of

*Unité de Recherche associée au Centre National de la Recherche Scientifique n° 1304

[†]Work supported by the French Research Project C³

transitions of the components, provided these transitions are allowed – or obliged – to be executed simultaneously. Arnold and Nivat [12, 3, 1] have named this construction, which is implemented in MEC, *synchronization product*.

Once a system of processes is represented as a transition system, one can extract, from this transition system, some informations about the behaviour of the system of processes it represents. It is what we call *analysis* of a transition system and it amounts to computing the set of states or the set of transitions which satisfy some property of interest when looking at the behaviour of the system. For instance it is easy to check if the transition system has “deadlocks”, i.e. states in which no transition is executable, or states in which every executable transition leads to a deadlock; a very simple algorithm can give the set of all these states. Thus an analyser is simply a tool which computes the set of all states or of all transitions of a given transition system satisfying some given property. The main feature of such an analyser is obviously the family of properties of states and transitions it can deal with.

In the systems Cesar [14] and emc [6], properties of states are expressed by formulas of branching time temporal logics. Given a formula F and a transition system \mathcal{A} , these systems compute the set $F_{\mathcal{A}}$ of states of \mathcal{A} satisfying F (or, at least, decide if the “initial state” of \mathcal{A} belongs to $F_{\mathcal{A}}$). In MEC we adopt a slightly different point of view, in some sense more algebraic than logical [7]. Let ω be some logical operator and let $F = \omega(F_1, \dots, F_n)$ be a formula. For a transition system \mathcal{A} , the set $F_{\mathcal{A}}$ of states satisfying \mathcal{A} depends on the sets $(F_i)_{\mathcal{A}}$ of states satisfying F_i , thus $F_{\mathcal{A}} = \omega_{\mathcal{A}}((F_1)_{\mathcal{A}}, \dots, (F_n)_{\mathcal{A}})$, where $\omega_{\mathcal{A}}$ is an operator defined on the cartesian product of the powerset of states with the powerset of states as a range. Then formulas can be considered as expressions which have to be evaluated, in a way very similar to what happens in programming languages with arithmetic or boolean expressions. Thus the language used in MEC to express properties consists in variables and constants ranging over the powerset of states and on the powerset of transitions, and of sorted operators; the basic mechanism implemented in MEC is the execution of assignments *variable* := *expression*, exactly like in programming languages.

It remains to define the basic operators which can be used to build expressions. We can take the operators of branching time temporal logics (which are computable in linear time with respect to the size of the transition system) but, also, any other kind of operator which is easily computable. For instance in MEC we use the operator which associates with a set of states the union of the strongly connected components intersecting this set; although this operator is not really a logical operator, it is as easy to compute as the other ones, because of the Tarjan’s algorithm [16] which is linear too.

Another feature of MEC is that the set of basic operators used to build expressions can be extended, in the same way that the set of operators in arithmetic expressions can be extended, in some programming languages, by defining new functions (especially recursive functions). It is well known that temporal logic operators can be characterized as least fixed points of equations [15, 8, 6], and this observation has led to the definition of the μ -calculus as an extension of branching time temporal logics [13, 11]. MEC provides for the definition of new operators characterized as least fixed points of systems of equations [7] and then its expressive power is at least as powerful as the expressive power of alternation-depth-one μ -calculus defined by Emerson and Lei [9]. Indeed these

new operators defined by systems of equations are still computable in linear time, like the basic ones, because of the Arnold-Crubillé's algorithm [2] to solve fixed point equations on transition systems in linear time.

1 Transition systems

1.1 Labelled transition systems

A *labelled transition system* over an alphabet A of *actions* or *events* is a tuple $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ where

- S is a finite set of *states*,
- T is a finite set of *transitions*,
- $\alpha, \beta : T \longrightarrow S$ are the mappings which associate with every transition t its *source state* $\alpha(t)$ and its *target state* $\beta(t)$,
- $\lambda : T \longrightarrow A$ labels a transition t by the action or event $\lambda(t)$ which causes this transition.

We assume that there never exist two different transitions with the same label between the same two states, i.e. the mapping $\langle \alpha, \lambda, \beta \rangle : T \longrightarrow S \times A \times S$ is injective.

1.2 Parametrized transition systems

A parametrized transition system is a labelled transition system given with some sets of designed states and some sets of designed transitions, called parameters. The role of these parameters is to give some additional informations on the transition system; it is the case when some states play a special role or when some transitions play a special role which is not specified by the label of the transition. Some example of such situations will be given below.

Example 1. Let us consider a boolean variable. It has two states, denoted by 0 and 1, according to the current value (0 or 1) of the variable. The set A of actions performed by such a boolean variable contains

- to0** which means that the variable is set to 0,
- to1** which means that the variable is set to 1,
- is0** which tests whether the value of the variable is 0,
- is1** which tests whether the value of the variable is 1,
- e** which does nothing.

The first two actions modify the value of the variable, i.e. its state, in an obvious way. The two tests can be executed only if the variable has the tested value, and this value is not modified. The last action, when executed, does not change the value of the variable. As we shall see later on, (example 3), this null action is a way to express the possibility of occurrence of events which does not modify the state of the variable.

Therefore the transition system has eight transitions: for each one of these transitions we give, in the following table, its source state, its target state, and its label.

transitions	source	target	label
t_1	0	0	e
t_2	0	0	to0
t_3	0	1	to1
t_4	0	0	is0
t_5	1	1	e
t_6	1	0	to0
t_7	1	1	to1
t_8	1	1	is1

Such transition systems often have a graphical representation, more readable when their size are little, as shown in figure 1

For the system MEC, transition systems are given in the form shown in figure 2. On the figure 2 one can notice the last line

`< initial = { 0 } >.`

which defines a parameter, named "initial", reduced to a single state, 0. This parameter is used to say that the state 0 has some special property, indeed it is the initial state of the transition system: the initial value of the variable, before any action is performed, is 0.

Example 2. Let us now consider the Peterson's algorithm for mutual exclusion of two processes. This algorithm uses three shared boolean variables, `flag[0]`, `flag[1]`, and `turn`, all three initialised to 0. Each one of the two processes executes the program given in figure 3 where `me` is equal to 0 and `other` is equal to 1 for the first process, and `me` is equal to 1 and `other` is equal to 0 for the second one. This program can also be represented by a transition system, states of which are the locations in the program and transitions between locations are labelled by elementary actions performed in the execution of the program. We also consider a special action `e` which does not change the state, which means that a process can stay idle at every moment. The MEC description of this transition system is given in figure 4.

In this transition system there are three state parameters:

`initial` which indicates the starting location,

`cs` which indicates the location where the process is in its critical section,

`ncs` which indicates the location where the process is not in its critical section.

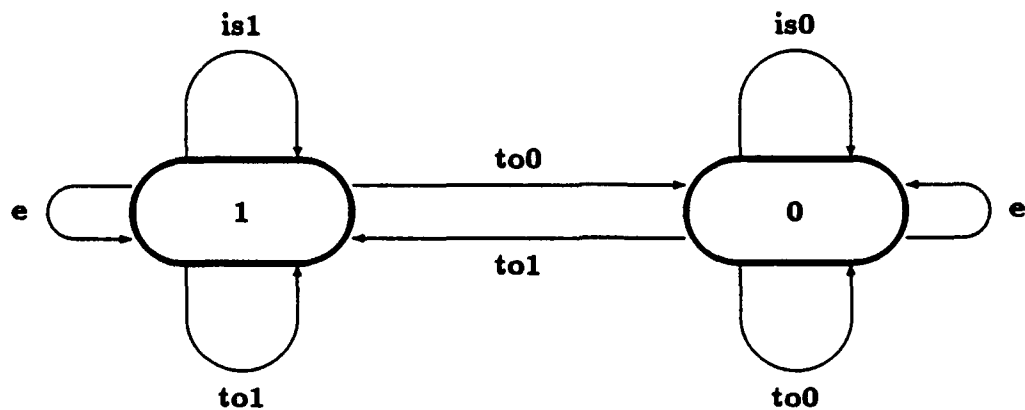


Figure 1: The graphical representation of the transition system for a boolean variable

```

transition_system b < width = 0 >;

0 |- e    -> 0 ,
    to0    -> 0 ,
    to1    -> 1 ,
    is0    -> 0 ;

1 |- e    -> 1 ,
    to0    -> 0 ,
    to1    -> 1 ,
    is1    -> 1 ;

< initial = { 0 } >.
  
```

Figure 2: The transition system for a boolean variable in MEC

```

proc( me , other ) =

while true do
begin
{NCS}      0: ... ;
{mutexbegin}  flag[me] := 1 ;
              1: turn = me ;
              2: WAIT (flag[other] = 0 OR turn = other) ;
{CS}       3: ... ;
{mutexend}  flag[me] := 0 ;
end

```

Figure 3: The Peterson algorithm

```

transition_system proc < width = 0 >;

0  |- e          -> 0 ,
    my_flag_to_1  -> 1 <property=(mb)>;

1  |- e          -> 1 <property=(mb)>,
    turn_to_me    -> 2 <property=(mb)>;

2  |- e          -> 2 <property=(mb)>,
    is_other_flag_0 -> 3 <property=(mb)>,
    is_turn_other  -> 3 <property=(mb)>;

3  |- e          -> 3 ,
    my_flag_to_0   -> 0 ;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.

```

Figure 4: The transition system for a process in MEC

There is also a transition parameter: it is the set of all transitions marked as having the property *mb*. These transitions are those executed by the processes when it tries to enter its critical section.

Indeed this transition system is obtained by interpreting the command

```
WAIT( ... OR ... )
```

in the following way: this command can be executed only if one of the two conditions is satisfied; in this case the execution of the process reaches the next location. It looks like idle waiting. Another interpretation of this command (busy waiting) could be: the process tests the first condition; if it is true it reaches the next location, otherwise it tests the second condition; if it is true it reaches the next location, otherwise it executes WAIT again. The transition system representing this interpretation is given in figure 5.

```
transition_system  procbis  < width = 0 >;

0    |- e                -> 0 ,
      my_flag_to_1        -> 1    <property=(mb)>;

1    |- e                -> 1    <property=(mb)>,
      turn_to_me          -> 2    <property=(mb)>;

2    |- e                -> 2    <property=(mb)>,
      is_other_flag_0     -> 3    <property=(mb)>,
      is_other_flag_1     -> 2bis <property=(mb)>;

2bis |- e                -> 2bis <property=(mb)>,
      is_turn_other       -> 3    <property=(mb)>,
      is_turn_me          -> 2    <property=(mb)>;

3    |- e                -> 3 ,
      my_flag_to_0        -> 0 ;

< initial = { 0 } ; cs = { 3 } ; ncs = { 0 } >.
```

Figure 5: Another transition system for a process in MEC

In the sequel we will deal only with the first of these two transition systems.

1.3 Paths

A *path* of length $n > 0$ in a transition system $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ is a sequence $p = t_1, \dots, t_n$ of transitions such that for all $i = 1, \dots, n-1$, $\beta(t_i) = \alpha(t_{i+1})$. The *source* of this path, denoted by $\alpha(p)$, and its *target*, denoted by $\beta(p)$ are respectively $\alpha(t_1)$ and $\beta(t_n)$. The *label* of this path, denoted by $\lambda(p)$, is the word $\lambda(t_1) \dots \lambda(t_n)$.

2 Synchronized systems

Let us consider n transition systems \mathcal{A}_i over the alphabets A_i of actions, for $i = 1, \dots, n$. Let us assume that these transition systems represent processes and shared objects constituting a system of interacting processes. (For simplicity, from now on, we shall also call processes the shared objects since they are also represented by transition systems). That means that some action in some process can be executed only simultaneously with some other action in some other process, or, on the opposite, cannot be executed simultaneously with some other action of some other process. Let us call *global action* a vector $\langle a_1, \dots, a_n \rangle$ where a_i belongs to A_i . Such a global action is executed when the actions a_i are simultaneously executed by the n processes. Thus the interactions between the processes of a system can be represented by the set of all global actions which are allowed to be executed and in [3], it is advocated that this kind of specification of the interactions between the processes of a system is general enough to formalise most of the concurrent systems of processes.

2.1 Synchronization constraints

As explained above, the interactions between the processes \mathcal{A}_i of a system are represented by a subset I of $A_1 \times \dots \times A_n$, called a *synchronization constraint*.

Example 3. Let us consider again Peterson's mutual exclusion algorithm for two processes. It is represented by a system containing two transition systems proc described in figure 4 and three boolean variables b described in figure 2. The second line of figure 6 gives the list of the transition systems of this system. The other lines are the elements of the synchronization constraint we are going to comment.

These elements are just those we get when obeying the following rules. (Here we temporarily come back to the distinction between processes and variables).

1. We assume this system runs on a single processor; therefore the two processes cannot execute simultaneously a non null action; moreover, we assume that this single processor is never idle and thus the two processes cannot be idle simultaneously.
2. Each action performed by a process consists in setting or testing a variable. When a process executes such an action, the corresponding variable executes the corresponding action and the other variables execute the null action.

as to rule 1, consider the first two columns of the array of figure 6; the first (resp. second) column contains all the actions performed by the first (resp. second) process. in each line of this subarray there is one and only one null action (denoted by e). as to rule 2, consider the first line of the array: when the first process sets its flag (i.e. `flag[0]`) to 0, then the first variable b , which represents `flag[0]`, executes the action "set to 0". in the third line it is the other process which sets its flag to 0 and it is the second variable, representing `flag[1]`, which executes the action "set to 0". in the same vein the actions `turn_to_me` are executed simultaneously with an action by the third variable representing `turn`, the value to which this variable is set depending on the process which sets the variable. The test `is_other_flag_0` is executed simultaneously with the action

is0, the boolean variable executing this action being dependent on the process which performs this test. Also the action is0 or is1 is executed by the third boolean variable simultaneously with the test is_turn_other executed by the second or first process.

```
synchronization_system peterson

< width = 5 ; list = (proc,proc,b,b,b) > ;

(my_flag_to_0 .e .to0 .e .e ) ;
(my_flag_to_1 .e .to1 .e .e ) ;
(e .my_flag_to_0 .e .to0 .e ) ;
(e .my_flag_to_1 .e .to1 .e ) ;
(turn_to_me .e .e .e .to0 ) ;
(e .turn_to_me .e .e .to1 ) ;
(is_other_flag_0 .e .e .is0 .e ) ;
(e .is_other_flag_0 .is0 .e .e ) ;
(is_turn_other .e .e .e .is1 ) ;
(e .is_turn_other .e .e .is0 ) .
```

Figure 6: The system representing Peterson's algorithm

2.2 Synchronized product

Given a vector $\langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ of transition systems, each $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \beta_i, \lambda_i \rangle$ over the alphabet A_i , the *free product* of $\langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ is the transition system $\langle S, T, \alpha, \beta, \lambda \rangle$ over $A_1 \times \dots \times A_n$ defined by

$$\begin{aligned} S &= S_1 \times \dots \times S_n, \\ T &= T_1 \times \dots \times T_n, \\ \alpha(t_1, \dots, t_n) &= \langle \alpha_1(t_1), \dots, \alpha_n(t_n) \rangle, \\ \beta(t_1, \dots, t_n) &= \langle \beta_1(t_1), \dots, \beta_n(t_n) \rangle, \\ \lambda(t_1, \dots, t_n) &= \langle \lambda_1(t_1), \dots, \lambda_n(t_n) \rangle. \end{aligned}$$

In some sense, the free product represents the evolution of the vector of transition systems when no constraint is set on the actions which can be performed simultaneously. In case of a synchronization constraint some transitions of this free product will never appear : those which are labelled by a vector of actions not allowed by the synchronization constraint. Hence we have the following definition.

Given a vector $\langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ of transition systems, each \mathcal{A}_i over the alphabet A_i , and a synchronization constraint I included in $A_1 \times \dots \times A_n$, the *synchronized product* of $\langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ with respect to I is the transition system $\langle S, T_I, \alpha, \beta, \lambda \rangle$ over $A_1 \times \dots \times A_n$ where

- $\langle S, T, \alpha, \beta, \lambda \rangle$ is the free product of $\mathcal{A}_1, \dots, \mathcal{A}_n$;
- T_I is the set of transitions $t = \langle t_1, \dots, t_n \rangle$ of T having their label $\lambda(t) = \langle \lambda_1(t_1), \dots, \lambda_n(t_n) \rangle$ in I .

Indeed the synchronized product computed by MEC is only a sub-transition system of the synchronized system defined above.

Each transition system $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \beta_i, \lambda_i \rangle$ which is a component of a synchronization system is assumed to have a parameter *initial*, as it is the case for the transition systems described in examples 1 and 2 (if this parameter is not mentioned, it is considered empty). This parameter defines a subset *initial_i* of S_i and the parameter *initial* of the product is defined as the subset $initial = initial_1 \times \dots \times initial_n$. The set of global states of the synchronized product of MEC is the set *Reach(initial)* of global states which can be reached from *initial*, i.e. the states of *initial* and the targets of paths having their sources in *initial*. The set of global transitions of the synchronized product of MEC is the set of global transitions having both their sources and their targets in *Reach(initial)*.

Example 4. The synchronized product, computed by MEC, of the synchronization system *peterson* given in figure 6 is given in figure 7. It was obtained by executing the MEC command `sync(peterson, res)`; where *res* is the name given to the product.

3 Elementary computations

The general form of a computation command in MEC is

variable := *expression*;

the *expression* is evaluated and its value is assigned to the *variable*. The value of an expression is either a set of states or a set of transitions.

3.1 Set variables

Variables used by MEC are of two different sorts according to the kind of set they can be assigned. A variable is implicitly declared when it appears for the first time in the left hand part of an assignment command and its sort is the sort of the expression (if the sort of the expression can be unambiguously determined, otherwise the assignment is rejected). Every declared variable is displayed on the terminal with the number of objects (states or transitions) of its value. Parameters are considered as variables with an initial value.

Let us remark also that variables are local to a transition system. If several transition systems are under examination, the same variable (properly speaking, the same variable name) can be used (like *initial* in the previous examples) but it will have different values according to the transition system with which it is associated (in other words a variable is a pair consisting in a variable name and a transition system).

```

transition_system res          < width =          5> ;
e(0.0.0.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.0) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.0) ;
e(1.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.0) ,
               (turn_to_me.e.e.e.to0) -> e(2.0.1.0.0) ;
e(2.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(2.1.1.1.0) ,
               (is_other fla.e.e.is0.e) -> e(3.0.1.0.0) ;
e(3.0.1.0.0) |- (e.my_flag_to_1.e.to1.e) -> e(3.1.1.1.0) ,
               (my_flag_to_0.e.to0.e.e) -> e(0.0.0.0.0) ;
e(0.1.0.1.0) |- (e.turn_to_me.e.e.to1) -> e(0.2.0.1.1) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.0) ;
e(1.1.1.1.0) |- (e.turn_to_me.e.e.to1) -> e(1.2.1.1.1) ,
               (turn_to_me.e.e.e.to0) -> e(2.1.1.1.0) ;
e(2.1.1.1.0) |- (e.turn_to_me.e.e.to1) -> e(2.2.1.1.1) ;
e(3.1.1.1.0) |- (e.turn_to_me.e.e.to1) -> e(3.2.1.1.1) ,
               (my_flag_to_0.e.to0.e.e) -> e(0.1.0.1.0) ;
e(2.2.1.1.0) |- (e.is_turn_othe.e.e.is0) -> e(2.3.1.1.0) ;
e(2.3.1.1.0) |- (e.my_flag_to_0.e.to0.e) -> e(2.0.1.0.0) ;
e(0.0.0.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(0.1.0.1.1) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.0.1.0.1) ;
e(1.0.1.0.1) |- (e.my_flag_to_1.e.to1.e) -> e(1.1.1.1.1) ,
               (turn_to_me.e.e.e.to0) -> e(2.0.1.0.0) ;
e(0.1.0.1.1) |- (e.turn_to_me.e.e.to1) -> e(0.2.0.1.1) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.1.1.1.1) ;
e(0.2.0.1.1) |- (e.is_other fla.is0.e.e) -> e(0.3.0.1.1) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.2.1.1.1) ;
e(0.3.0.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(0.0.0.0.1) ,
               (my_flag_to_1.e.to1.e.e) -> e(1.3.1.1.1) ;
e(1.1.1.1.1) |- (e.turn_to_me.e.e.to1) -> e(1.2.1.1.1) ,
               (turn_to_me.e.e.e.to0) -> e(2.1.1.1.0) ;
e(1.2.1.1.1) |- (turn_to_me.e.e.e.to0) -> e(2.2.1.1.0) ;
e(2.2.1.1.1) |- (is_turn_othe.e.e.e.is1) -> e(3.2.1.1.1) ;
e(3.2.1.1.1) |- (my_flag_to_0.e.to0.e.e) -> e(0.2.0.1.1) ;
[4~e(1.3.1.1.1) |- (e.my_flag_to_0.e.to0.e) -> e(1.0.1.0.1) ,
               (turn_to_me.e.e.e.to0) -> e(2.3.1.1.0);
< initial = { e(0.0.0.0.0) } >.

```

Figure 7: A synchronized product

3.2 Expressions

Expressions are built up from variables and operators. Among these operators are set-theoretical (or boolean) operators union, intersection, and difference as well as the constants "empty", denoted by $\{\}$, and "all", denoted by $*$. Of course the use of these operators in expressions are submitted to sort restrictions.

Some others operators are primitive and will be described below. New operators can be defined by the users and this will be explained in the next section.

Finally there are some other ways to define sets of states and sets of transitions. We will not list all these ways here and we refer to the user manual [4].

Example 5. Let us consider the transition system **res** of figure 4 constructed by MEC, which will be our running example from now on.

First of all we want to know if the *mutual exclusion* property is verified, i.e. if the two processes can be or not both together in their critical section. Let us remind that the set of states in which a process is in its critical section is defined by the parameter **cs** as shown in figure 2. Therefore we have just to know whether there are (global) states in which

- (i) the first component is in the parameter **cs** of the first transition system of the synchronization system **peterson**,
- (ii) the second component is in the parameter **cs** of the second transition system of the synchronization system **peterson**.

The sets of states satisfying (i) and (ii) are respectively denoted by **cs[1]** and **cs[2]**; hence the set of states not satisfying the mutual exclusion property is defined and/or computed by the assignment **nok** := **cs[1] /\ cs[2]**;

After execution of this command, it appears on the screen that the value of **nok** is a set of states which has 0 element. □

3.3 Primitive operators

Let us denote by σ and τ the sorts "set of states" and "set of transitions". We can use the following operators **src** of sort $\tau \rightarrow \sigma$, **tgt** of sort $\tau \rightarrow \sigma$, **rsrc** of sort $\sigma \rightarrow \tau$, **rtgt** of sort $\sigma \rightarrow \tau$. The interpretation of these operators is as follows, for a given transition system $\langle S, T, \alpha, \beta, \lambda \rangle$: if Q is a set of states included in S and R a set of transitions included in T , then

$$\begin{aligned}\mathbf{src}(R) &= \{\alpha(t) \mid t \in R\}, \\ \mathbf{tgt}(R) &= \{\beta(t) \mid t \in R\}, \\ \mathbf{rsrc}(Q) &= \{t \mid \alpha(t) \in Q\}, \\ \mathbf{rtgt}(Q) &= \{t \mid \beta(t) \in Q\}.\end{aligned}$$

In other words **src**(R) and **tgt**(R) are respectively the sets of sources and targets of transitions in R and **rsrc**(Q) and **rtgt**(Q) are their reciprocals : the sets of transitions having their source and their target in Q .

Example 6. If Q is a set of states, the set $Pred(Q)$ is the set of all states which are the source of a transition whose target is in Q . If Q is a variable whose value is Q , $Pred(Q)$ is the value of the expression $src(rtgt(Q))$.

In a similar way $Succ(Q)$ is the value of the expression $tgt(rsrc(Q))$.

If R denotes a set R of transitions, the expression $src(R/\rtgt(Q))$ evaluates to the set of sources of transitions in R having their target in Q and $tgt(R/\rsrc(Q))$ to the set of targets of transitions in R having their source in Q . \square

We also use the binary operator $loop$ of sort $\tau\tau \rightarrow \tau$ defined by $t \in loop(R, R')$ if and only if t belongs to a path p such that

- (i) the source of p is equal to its target,
- (ii) every transition of p is in R' ,
- (iii) some transition of p is in R ,

In other words, a transition t is in $loop(R, R')$ if it belongs to some loop in R' containing some transition in R .

Example 7. Let us now look for a *livelock* in the transition system **res**.

Roughly speaking there is a livelock if there is an infinite execution where

- (i) both processes are always in their "mutexbegin", and
- (ii) none of the processes remains "inactive" forever.

Condition (i) means that both processes are trying to enter their critical section and never succeed; condition (ii) means that both processes are really trying to enter their critical section : if one of the processes stays idle forever surely it will not enter its critical section and could even prevent the other process to enter.

Since the only waiting action is denoted by e , a process is active during a transition t if the corresponding component of the label of this transition is not equal to e . Therefore the sets of transitions in which the first and the second processes are active are computed by

active1 := !label[1] # "e"; and

active2 := !label[2] # "e";

The set of transitions in which both processes are in their "mutexbegin" is computed by

ll := mb[1]/\mb[2];

If there is a livelock, there is an infinite path p such that

- (ia) all its transitions are in **ll**;
- (iia) it has an infinite number of transitions in **active1**;
- (iiia) it has an infinite number of transitions in **active2**.

Since there is a finite number of states this infinite path p contains a loop p' which satisfies

- (ib) all its transitions are in $l1$;
- (iib) it has at least one transition in $active1$;
- (iiib) it has at least one transition in $active2$.

Conversely if there is a loop satisfying (ib,iib,iiib), there exists an infinite path satisfying (ia,iaa,iaaa). Hence there is a livelock if and only if there is a loop satisfying (ib,iib,iiib).

The set $l10$ of transitions belonging to a loop satisfying (ib) is computed by
 $l10 := loop(*,l1);$
The set $l11$ of transitions belonging to a loop satisfying (ib) and (iib) is computed by
 $l11 := loop(active1,l10);$
(but also by $l11 := loop(active1,l1);$
Finally the set $l12$ of transitions belonging to a loop satisfying (ib),(iib) and (iiib) is computed by
 $l12 := loop(active2,l11);$

Here again this set is empty. □

4 The definition of new operators

4.1 Preliminary examples

Let us consider some given transition system \mathcal{A} .

1. Let us consider the set $Reach(initial)$ which was used in the definition of the synchronized product (cf. 1.5). Indeed for every set Q of states one can define the set $Reach(Q)$ containing Q and the targets of paths having their source in Q . Thus $Reach$ can be considered as an operator of sort $\sigma \rightarrow \sigma$ and one can think of adding it to the primitive operators.

But we can also remark that this operator can be formally defined in the following way.

Let us consider the operator $Succ$ defined in example 6. We have the following equality:

$$Reach(Q) = Q \cup Succ(Reach(Q)) \quad (1)$$

By definition $Q \subset Reach(Q)$, and if there is a path from a state s to a state s' , there is also a path from s to every state of $Succ(\{s'\})$, thus $Succ(Reach(Q)) \subset Reach(Q)$. Conversely let $Q_0 = Q$ and Q_n , for $n > 0$, be the set of targets of paths of length n having their source in Q ; then it is clear that $Reach(Q) = \bigcup_{n \geq 0} Q_n$ and that $Q_{n+1} = Succ(Q_n)$, hence $Reach(Q) = Q_0 \cup \bigcup_{n \geq 0} Succ(Q_n) \subset Q \cup Succ(Reach(Q))$.

We have even more: not only $Reach(Q)$ is a solution of the equation

$$X = Q \cup Succ(X) \quad (2)$$

but it is the least set of states (for inclusion) satisfying this equation: if X is a set of states satisfying the equation 2 then $Q_0 = Q \subset X$ and if $Q_n \subset X$ then $Q_{n+1} = Succ(Q_n) \subset Succ(X) \subset X$.

Therefore we can give $Reach$ the following definition: for every set Q of states, $Reach(Q)$ is the least solution of 2. This definition is expressed in MEC by

```
function reach(Q:state) return X:state;
begin
X = Q \/ tgt(rsrc(X))
end.
```

Once this function is defined, the operator $reach$ can be used in expressions exactly like primitive operators; its sort is $\sigma \rightarrow \sigma$, as expressed by the first line of the definition.

2. Let us now consider the two sets $ReachEven(Q)$ and $ReachOdd(Q)$ of states reachable from a state of Q by a path of even or of odd length. We have

$$ReachEven(Q) = \bigcup_{n \geq 0} Q_{2n}$$

$$ReachOdd(Q) = \bigcup_{n \geq 0} Q_{2n+1}$$

where the sets Q_n are defined above. Then

$$ReachEven(Q) = Q \cup Succ(ReachOdd(Q))$$

$$ReachOdd(Q) = Succ(ReachEven(Q))$$

and the pair $\langle ReachEven(Q), ReachOdd(Q) \rangle$ is the least pair $\langle X, Y \rangle$ of sets of states satisfying

$$X = Q \cup Succ(Y)$$

$$Y = Succ(X)$$

Thus we can define the operator $reacheven$ of sort $\sigma \rightarrow \sigma$ by

```
function reacheven(Q:state) return X:state;
var Y:state
begin
X = Q \/ tgt(rsrc(Y));
Y = tgt(src(X))
end.
```

It could be noticed that the auxiliary variable Y is not really useful since this definition is equivalent to


```

function reacheven(Q:state) return X:state;
begin
X = Q \/ tgt(rsrc(tgt(src(X))));
end.

```

Here is an example where the definition cannot be simplified this way.

3. Let us consider some set R of transitions; we denote by $F(R, Q)$ (resp. $G(R, Q)$) the set of states reachable from Q by a path containing an even (resp. odd) number of transitions in R . Let us remark that for any set Z_t of transitions and any set Z_s of states, the set $\text{tgt}(Z_t \cap \text{rsrc}(Z_s))$, denoted by $\text{Succ}(Z_t, Z_s)$, is the set of states reachable from Z_s by one transition in Z_t . Thus $\langle F(R, Q), G(R, Q) \rangle$ is the least pair $\langle X, Y \rangle$ satisfying

$$\begin{aligned}
X &= Q \cup \text{Succ}(* - R, X) \cup \text{Succ}(R, Y) \\
Y &= \text{Succ}(* - R, Y) \cup \text{Succ}(R, X)
\end{aligned}$$

where $* - R$ is the complement of R . This definition is expressed in MEC by

```

function F(R:trans , Q:state) return X:state;
var Y :state
begin
X = Q \/ tgt((* - R)/\rsrc(X)) \/ tgt((R/\rsrc(Y));
Y = tgt((* - R)/\rsrc(Y)) \/ tgt((R/\rsrc(X))
end.

```

4.2 Systems of equations

We are now going to formally define the systems of equations which can be used to define new operators in MEC.

Basic operators Let D be the heterogeneous algebraic signature with two sorts, σ and τ , containing the following operators [7] :

- $0_\sigma, 1_\sigma, 0_\tau, 1_\tau$: constants of sorts σ and τ ;
- $\cup_\sigma, \cap_\sigma, -_\sigma$: binary operators of sort $\sigma\sigma \rightarrow \sigma$;
- $\cup_\tau, \cap_\tau, -_\tau$: binary operators of sort $\tau\tau \rightarrow \tau$;
- src, tgt : unary operators of sort $\tau \rightarrow \sigma$;
- rsrc, rtgt : unary operators of sort $\sigma \rightarrow \tau$.

If $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$ is a transition system, it is given a D -structure in the following way :

- The set of elements of sort σ (resp. τ) is $\wp(S)$ (resp. $\wp(T)$).

- The interpretation of the operators is

$$\begin{aligned}
(0_\sigma)_A &= \emptyset, \\
(1_\sigma)_A &= S, \\
(0_\tau)_A &= \emptyset, \\
(1_\tau)_A &= T, \\
P(\cup_\sigma)_A P' &= P \cup P', \\
P(\cap_\sigma)_A P' &= P \cap P', \\
P(-_\sigma)_A P' &= P - P', \\
R(\cup_\tau)_A R' &= R \cup R', \\
R(\cap_\tau)_A R' &= R \cap R', \\
R(-_\tau)_A R' &= R - R', \\
src_A(R) &= \{\alpha(t) | t \in R\}, \\
tgt_A(R) &= \{\beta(t) | t \in R\}, \\
rsrc_A(P) &= \{t | \alpha(t) \in P\}, \\
rtgt_A(P) &= \{t | \beta(t) \in P\}.
\end{aligned}$$

All these operators, but $-_\sigma$ and $-_\tau$, have monotonic interpretations with respect to set inclusion.

Signed terms Let $X_n = \{x_1, \dots, x_n\}$ and $Y_m = \{y_1, \dots, y_m\}$ be two sets of variables of sort σ and τ . We can build terms with these variables and the operators of D . If t is such a term, its interpretation t_A will be a mapping from $\wp(S)^n \times \wp(T)^m$ into $\wp(S)$ or $\wp(T)$ according to the sort of this term.

Since the interpretation of a difference operator is not monotonic, the interpretation of a term is not necessarily monotonic. However we can consider that the interpretation of a difference becomes monotonic if its first argument is ordered by inclusion and its second argument is ordered by the inverse relation : containment.

This led us to consider two kinds of order on $\wp(S)$ and $\wp(T)$, inclusion and containment. We shall denote these powersets by $\wp^+(S)$ and $\wp^+(T)$ (resp. $\wp^-(S)$ and $\wp^-(T)$) when we wish to make clear that they are ordered by inclusion (resp. containment). For each sort, we consider two kinds of variables : positive variables ranging over a powerset ordered by inclusion and negative variables ranging over a powerset ordered by containment. Let X^+ and X^- be two sets of positive and negative variables of sort σ , Y^+ and Y^- two sets of positive and negative variables of sort τ and Z_σ et Z_τ two set of "parameters", which will be interpreted as arbitrary sets.

We inductively define the sets of positive and negative terms of sort σ , T_σ^+ et T_σ^- , and of positive and negative terms of τ , T_τ^+ and T_τ^- by

- $X^+ \subset T_\sigma^+$, $X^- \subset T_\sigma^-$, $Y^+ \subset T_\tau^+$, $Y^- \subset T_\tau^-$;
- $\{0_\rho, 1_\rho\} \cup Z_\rho \subset T_\rho^+ \cap T_\rho^-$; ($\rho = \sigma, \tau$);

- if t_1 and t_2 belong to T_ρ^ς then $t_1 \cup_\rho t_2$, $t_1 \cap_\rho t_2$ belong to T_ρ^ς ; ($\rho = \sigma, \tau$; $\varsigma = +, -$);
- if t belongs to T_τ^ς then $src(t)$ and $tgt(t)$ belong to T_σ^ς ; ($\varsigma = +, -$);
- if t belongs to T_σ^ς then $rsrc(t)$ and $rtgt(t)$ belong to T_τ^ς ; ($\varsigma = +, -$);
- if t_1 belongs to T_ρ^ς and t_2 belongs to $T_\rho^{\varsigma'}$ then $t_1 -_\rho t_2$ belongs to T_ρ^ς ; ($\rho = \sigma, \tau$; $\langle \varsigma, \varsigma' \rangle = \langle +, - \rangle, \langle -, + \rangle$);

If t is a term of T_ρ^ς its interpretation t_A is then monotonic or antimonotonic (according to the value of ς) when the values of variables in Z are fixed and values of other variables are ordered according to their sign.

Example 8. If Z_τ is a parameter of sort τ , Z_σ a parameter of sort σ , X_+ a positive variable of sort σ and Y_- a negative variable of sort τ , then $Z_\tau \cap_\tau rtgt(1_\sigma -_\sigma X_+)$ is a negative term of sort τ and $Z_\sigma \cup_\sigma (1_\tau -_\tau src(Y_-))$ is a positive term of sort σ . \square

Systems of equations Let us consider the following sets of variables :

$$\begin{aligned} X^+ &= \{X_1^+, \dots, X_n^+\}, \\ X^- &= \{X_1^-, \dots, X_{n'}^-\}, \\ Y^+ &= \{Y_1^+, \dots, Y_m^+\}, \\ Y^- &= \{Y_1^-, \dots, Y_{m'}^-\}, \\ Z_\sigma &= \{Z_1, \dots, Z_p\}, \\ Z_\tau &= \{Z'_1, \dots, Z'_{p'}\}, \end{aligned}$$

We consider the sets of terms $T_\sigma^+, T_\sigma^-, T_\tau^+, T_\tau^-$ built with these variables.

A system of equations Σ is

$$\begin{aligned} \{X_i^+ &= u_i^+ | 1 \leq i \leq n\} \cup \\ \{X_i^- &= u_i^- | 1 \leq i \leq n'\} \cup \\ \{Y_i^+ &= v_i^+ | 1 \leq i \leq m\} \cup \\ \{Y_i^- &= v_i^- | 1 \leq i \leq m'\}. \end{aligned}$$

where $u_i^+ \in T_\sigma^+$, $u_i^- \in T_\sigma^-$, $v_i^+ \in T_\tau^+$, $v_i^- \in T_\tau^-$.

With a system of equations Σ and a transition system \mathcal{A} we associate the ordered set

$$D_1 = \wp^+(S)^n \times \wp^-(S)^{n'} \times \wp^+(T)^m \times \wp^-(T)^{m'}$$

and the set

$$D_2 = \wp(S)^p \times \wp(T)^{p'}$$

ordered by the empty order. Then Σ defines a mapping

$$\Sigma_{\mathcal{A}} : D_1 \times D_2 \longrightarrow D_1$$

This mapping is monotonic with respect to the order defined componentwise on D_1 and $D_1 \times D_2$. Thus it has a least fixed point $\mu\Sigma_{\mathcal{A}} : D_2 \longrightarrow D_1$.

Now if we choose one of the signed variables, by composing $\mu\Sigma_{\mathcal{A}}$ with the projection of D_1 on its component associated with this variable, we get a mapping from D_2 in $\wp(S)$ or $\wp(T)$, according to the sort of the variable. Thus we can assume that a new operator is defined by :

- the list of sorted parameters,
- the list of sorted and signed variables,
- the selected variable defining the result,
- the list of equations, one for each variables.

The interpretation of such an operator in any transition system will be the mapping defined above.

Example 9. Let us consider the two terms of the example 8. The parameters they contains are Z_τ and Z_σ , and the variables are X_+ and Y_- . Let us consider the two equations

$$\begin{aligned} X_+ &= Z_\sigma \cup_\sigma (1_\tau -_\tau \text{src}(Y_-)) \\ Y_- &= Z_\tau \cap_\tau \text{rtgt}(1_\sigma -_\sigma X_+) \end{aligned}$$

For every transition system \mathcal{A} this defines a mapping from $\wp(T) \times \wp(S)$ in $\wp(S) \times \wp(T)$; if we choose X_+ as principal variable we get a mapping from $\wp(T) \times \wp(S)$ in $\wp(S)$.

Let us call **unavoidable** this operator for some reasons which will be explained below. In MEC its definition will be written

```
function unavoidable(Zt:trans ; Zs:state) return X:state;
var Y:_trans
begin
X = Zs \ / (* - src(Y));
Y = Zt /\ rtgt(* - X)
end.
```

Let us remark that negative variables are specified by an "underscore" preceding their sort.

The name "unavoidable" given to this operator comes from the following property.

Let \mathcal{A} be any transition system, Q some set of states and R some set of transitions. Then a state s belongs to $\text{unavoidable}(R, Q)$ if and only if every maximal path in \mathcal{A} (i.e. an infinite path or a finite path whose last state has no successor in \mathcal{A}) originated in s and containing only transitions in R contains a state in Q .

To prove this assertion let us remark that $\text{unavoidable}(R, Q)$ is also the least solution of $X = Q \cup (* - \text{Pred}(R, * - X))$ where $\text{Pred}(A, B) = \text{src}(A \cap \text{rtgt}(B))$ is the set of origins of transition of A having their target in B . Thus $s \in X$ if and only if $s \in Q$ or all transitions of R of source s have their target in X .

In particular $\text{unavoidable}(T, \emptyset)$ is the set of states s such that every maximal path originated in s is finite. This can be considered as a definition of "deadlocking" states, since once in such a state it is impossible to start an infinite computation.

Least and greatest fixed points The use of signed variables allows to define new operators as greatest fixed point of system of equations as well as least fixed point just by playing on the signs : the least element for inclusion is the greatest one for containment and vice-versa!

Computation of least fixed points If an expression contains an operator defined by a system of equation it remains to evaluate this expression. This amounts to computing the value of $\text{op}(Z_1, \dots, Z_n)$ when the values of Z_1, \dots, Z_n are known and thus to computing the least fixed point of the equations defining op when the parameters occurring in these equations are given. This can be done in time linear with respect to the size of the transition system (i.e. number of states and number of transitions, using an algorithm described in [2]). Thus all computations performed by MEC are done in a time linear with the size of the transition system.

5 An example of use

MEC has been used to check some mutual exclusion algorithms. It allowed to discover that Burns's algorithms [5] contained livelocks in the case of four processes.

The transition system obtained by synchronizing four processes, four boolean flags and a "turn" variable, representing the symmetrical Burns's algorithm with four processes has 65 016 states, 260 064 transitions stored in about 13 Mbytes of memory. On a Sun 3/60, it takes 20 minutes of CPU to construct this product and 11 minutes to compute unavoidable using the linear algorithm of Arnold and Crubillé.

References

- [1] A. Arnold. Transition systems and concurrent processes. In *Mathematical problems in Computation theory (Banach Center Publications, vol. 21)*, 1987.
- [2] A. Arnold and P. Crubillé. A linear algorithm to solve fixed point equations on transition systems. *Inf. Process. Lett.*, 29:57–66, 1988.
- [3] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'Informatique"*, pages 35–68, 1982.
- [4] D. Bégay. *Mode d'emploi MEC*. Technical Report I-8915, Université Bordeaux I, 1989.
- [5] J. E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Annual Symp. on Foundations of Computer Science*, pages 169–174, 1981.
- [6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logics specifications. *ACM Trans. Prog. Lang. Syst.*, 8:244–263, 1986.
- [7] A. Dicky. An algebraic and algorithmic method for analysing transition systems. *Theoretical Comput. Sci.*, 46:285–303, 1986.
- [8] E. A. Emerson and E. C. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. de Bakker and J. van Leeuwen, editors, *7th Int. Coll. on Automata, Languages and Programming*, pages 169–181, Lect. Notes. Comput. Sci. 85, 1980.
- [9] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ -calculus. In *Symp. on Logic in Comput. Sci.*, pages 267–278, 1986.
- [10] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19:371–384, 1976.
- [11] D. Kozen. Results on the propositional μ -calculus. *Theoretical Comput. Sci.*, 27:333–354, 1983.
- [12] M. Nivat. Sur la synchronisation des processus. *Revue Technique Thomson-CSF*, 11:899–919, 1979.
- [13] V. Pratt. A decidable μ -calculus. In *Proc. 22nd Symp. on Foundations of Comput. Sci.*, pages 421–427, 1981.
- [14] J.-P. Queille. *Le système CESAR: Description, spécification et analyse des applications réparties*. PhD thesis, I.N.P., Grenoble, 1982.
- [15] J. Sifakis. *Global and local invariants in transition systems*. Technical Report 274, IMAG, Grenoble, 1981.
- [16] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

ALGEBRAIC SEMANTICS OF REAL-TIME PROCESS SPECIFICATIONS

S. KAPLAN* **, G. DEUTSCH*

* Laboratoire de Recherche en Informatique, U.A. 410 du C.N.R.S
Université Paris-Sud, Bât. 490, 91405 ORSAY Cedex, FRANCE

** Department of Computer Science
Bar-Ilan University, 52100 Ramat-Gan, ISRAEL

Abstract :

The article presents the new formalism of *real-time process specifications*, that allow to describe concurrent systems within the algebraic specification approach. Several real-time aspects are taken into consideration. Atomic actions receive a duration, and each of them may be preceded by an arbitrary system overhead (smaller than a given constant). They may be executed concurrently, provided no dynamic access conflict is generated. In the latter case, one of them is automatically postponed. Wait primitives are introduced, together with guarded commands. Emphasis is put on methodological aspects, illustrated by an example, and on semantics questions.

1 Introduction

Formal specifications are widely considered as crucial in systematic software development. Their essential purpose is to state a precise and non-ambiguous definition of the problem under consideration, allowing to check the adequacy of later realizations. Among several approaches, *algebraic specifications* have been quite successful, due to the large level of abstraction that they authorize and to their hierarchical aspect. Classically, algebraic specification only allow for the description of purely sequential systems. Recently, the author – together with A. Pnueli – came up with an extension of the theory, called *process specifications*, that take into account *concurrent systems* [KP 87], [Kaplan 88], [Kaplan 89]. During the same period, different authors suggested other approaches, within the algebraic approach.

Our *process specifications* constitute at the same time a mathematical semantics for concurrency, and a *methodology* in order to develop specifications in a systematical fashion, prove properties about them, establish the correctness of possible implementations, etc. An idealization is made w.r.t. the reality, namely that atomic actions are points in the time, without duration (as in CCS [Milner 80], CSP [Hoare 78], MEIJE [Boudol 84], ACP [BK 84]). Also, semantics of the parallelism is described by interleavings.

In this article, we extend the mathematical theory and the methodological aspects of our process specifications to *real-time issues* (cf. e.g. [Wirth 77, LL 83, KSR 88, Da 85, JM 86], etc.). In our extended approach, atomic actions receive a duration, several atomic actions may take place simultaneously (provided that no conflicts arise), and guarded commands and *wait* primitives are introduced. Moreover, we take into account the fact that each action may be preceded by an arbitrary delay, ranging from 0 to a constant α , in order to simulate system overheads.

It turns out that the mathematical treatment of parallelism (i.e. its definition by means of equations) becomes more delicate; particular emphasis is put on this point in the article.

Section 2 introduces the general background, and defines *real-time process specifications*. Sections 3 and 4 define the semantics of a specifications, firstly informally and then formally. Several notions are discussed, including observational aspects, an implementation relation between specifications, etc. Finally, several extensions are discussed in Section 5. Throughout the paper, some aspects have been simplified for sake of clarity; we refer to [Deutsch 91] for a complete technical treatment. The reader is assumed to have general knowledge about algebraic specifications (cf. [Wirsing 91] for a recent survey); no prior acquaintance about our process specifications is required.

Notations : through this article, we use the notion of *record*, denoted by $\{token_1, \dots, token_n\}$, or by $\langle field_1, token_1 \rangle, \dots, \langle field_n, token_n \rangle$. We denote by $r[A]$ the content of a record r at field A . $r[A \setminus exp]$ stands for r where the token at A is replaced by the expression exp (it is assumed that type-checking is performed). This notation is also used if A is a set of fields; in this case exp is generally itself a record, which we restrict to A . If $A \cap B = \emptyset$, we also use the notation $r[A \setminus exp, B \setminus exp']$ with obvious meaning. Sub-intervals of the real line are denoted by $[a..b]$, $(a..b]$, etc. In what follows, letters p, q, \dots stand for processes (defined hereafter), a, b, \dots for atomic actions, $\alpha, \varepsilon, \tau$ for real positive numbers, s for a state, and t for a time.

2 Real-time Process Specifications

In this section, we first explain informally how *real-time process specifications* work. We then provide syntax for them, illustrated on an example. Semantics issues are treated in the next sections.

In our approach, the world is hierarchized into two layers: *processes* and *environment*. A process applies to a given environment and produces a set of possible new environments. More precisely:

- The *environment* is a record with two fields:
 - The *time* component represents the current time of the system, given by an internal clock. Time is assumed to be isomorphic to the real half-line $[0.. \infty)$ (cf. Appendix A for a discussion).
 - The *state* component represents the current state of the system. It is actually a classical (i.e. "sequential") algebraic specification, written hierarchically (cf. e.g. [Wirsing 91]). Such specifications are written in the PLUSS language [Gaudel 85], [Bidoit 89]. Mostly often, a state is itself a *record*.

An environment is denoted by $\langle state, s \rangle, \langle time, t \rangle$ — or simply by $\{s, t\}$ when no confusion arises. The result of a computation is actually a *set* of environments, which models nondeterminism. Sets are expressed with the two operators " \emptyset " and " \cup ", with their usual meaning.

- The *processes* represent the (possibly concurrent) actors that apply on the environment. Processes are built out of basic entities called *actions*. In turn, they are of two kinds:
 - *atomic actions* that are un-interruptible. They are characterized by how they affect the current state of the environment and the current time.
 - *compound actions*, that are defined by composition of smaller actions.

Processes are composed via the following operations: nondeterministic choice (“+”), sequential composition (“;”), and parallel composition (“||”). Wait actions and guarded commands are also available — their presentation is postponed to section 5 for sake of readability.

Now, a specifier defines a real-time process specification according to the syntax of Figure 1. The

```

process spec NAIVE-ADDER =
  state : [ < x, NAT >, < y, NAT >, < z, NAT > ]
  atomic actions :
    add _ to _ : z/y/z x z/y/z → atomic
  compound action :
    add _ to _ and _ : z/y/z x z/y/z x z/y/z → compound
  application of the atomic actions to the state :
    add w to w' : s = s[w' \ s[w'] + s[w']]
  duration of atomic actions :
    μ(add w to w', s) = (s[w'] + s[w'] + 1) × 10-3
  definition of the compound action :
    add w to w' and w'' = add w to w' || add w to w''
  var : s : state, w, w', w'' : z/y/z
end NAIVE-ADDER

```

Figure 1: An example of *real-time process specification*

sort $x/y/z$, stands for the enumerated type {“x”, “y”, “z”}. *NAT* is a classical specification for natural numbers. an expression $a : s$ stands for the (static) application of the atomic action a to the state s . Similarly, $\mu(a, s)$ stands for the duration of the application of a to s .

The application of an atomic action a to a *full environment* [$\langle \text{state}, s \rangle, \langle \text{time}, t \rangle$] is therefore described by the following equation :

$$a :: [\langle \text{state}, s \rangle, \langle \text{time}, t \rangle] = [\langle \text{state}, a : s \rangle, \langle \text{time}, t + \mu(a, s) \rangle] \quad (1)$$

In general, a user may now relate to expressions s.a. $p :: [s, t]$, where p : *process* is a process made of the above composition operators and of the actions — describing this way a possibly concurrent computation. Thanks to the semantics equations given in the next sections, this last expression is provably equal to some $\bigcup_{i=1}^N [\langle \text{state}, s_i \rangle, \langle \text{time}, [t_i..t'_i] \rangle]$ (cf. Theorem 2) — with some restrictions on p , however.

3 Informal Presentation of the Semantics

A *real-time process specification* \mathcal{P} may be seen – semantically speaking – as a *macro* that expands into a *classical, algebraic specification*, which we denote by $\text{Sem}(\mathcal{P})$. The hierarchical organization of $\text{Sem}(\mathcal{P})$ is shown in Figure 2 (where thick arrows are constructors, and thin arrows are derived operators). The semantics of \mathcal{P} is then, by definition, the *classical semantics* of $\text{Sem}(\mathcal{P})$, i.e. the

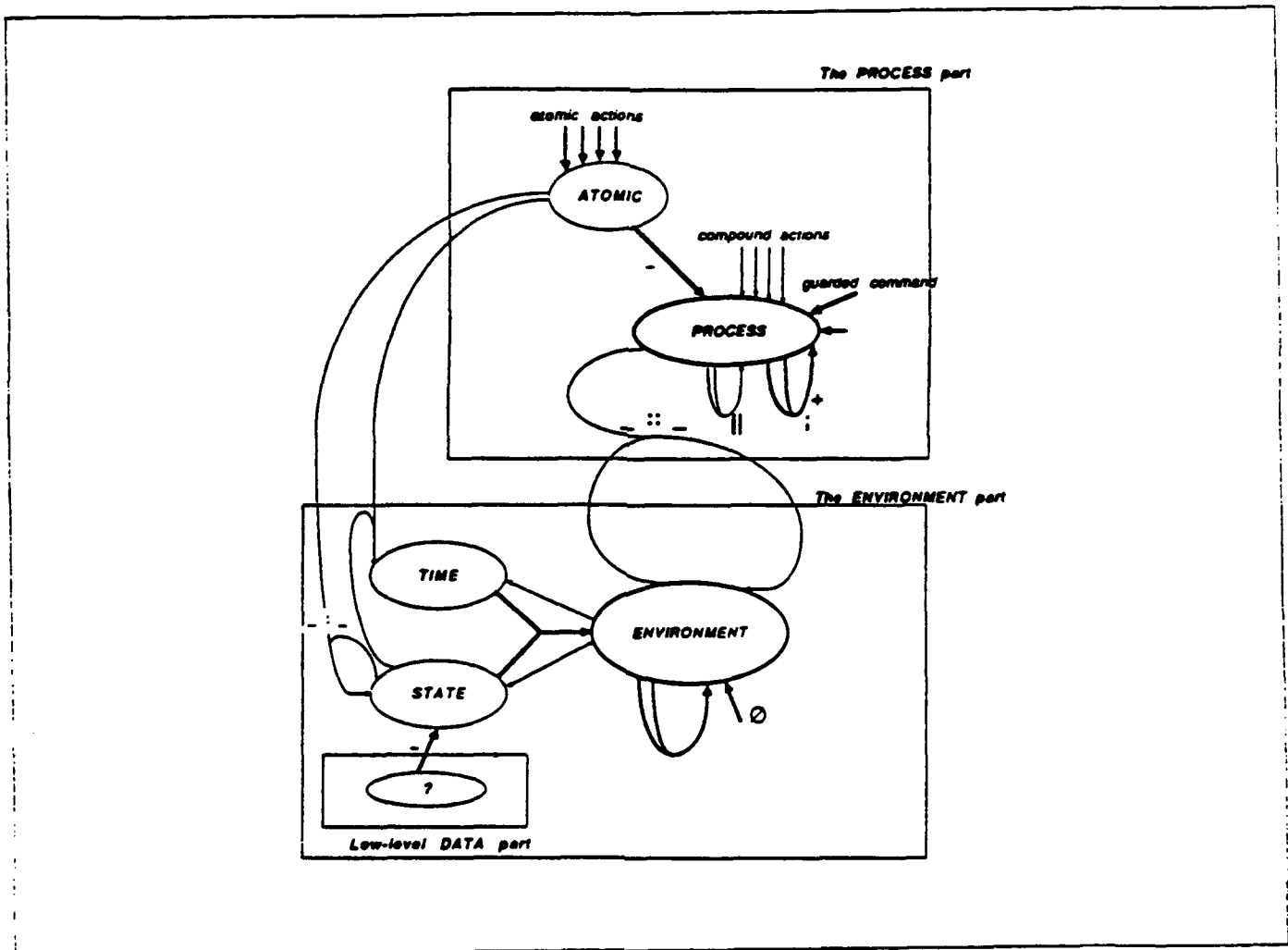


Figure 2: Hierarchical organization of a *real-time process specification*

class of its hierarchical models. The models considered here are standard, except that the sort *time* is isomorphic to the real half-line $[0.. + \infty)$ (cf. Appendix A for a discussion). In this section, we proceed with explaining informally the semantics of our specifications. Formal presentation appears in Section 4.

We assume that before engaging itself, each action may be delayed for a time $\epsilon \in [0, \alpha]$, where α is a constant that determines the maximal system overhead.

Call *writes* of an action a (denoted by $Write(a)$) the fields of the state that are modified by a , and *reads* of a (denoted by $Read(a)$) the fields of the state that are used when applying a . In the most common cases, these two notions can be determined syntactically by looking at the equations defining the application of a .¹ We now have the essential constraint:

Two actions cannot take place simultaneously if their writes intersect.

Suppose that an action writing on a given field is taking place, and that another action writing on the same field is ready to start (such a situation is called a *dynamic conflict*). In such a case, the latter is delayed (possibly longer than the maximal overhead) until the former terminates. On the other hand, simultaneous reads are admissible.

¹Consider such an equation of the form $a : s = e$. If $s[f \setminus \dots]$ appears as a sub-expression of e , then $f \in Write(a)$. Likewise, if e contains a subexpression $s[f]$, then $f \in Read(a)$.

When an action starts, the value of its “reads” is stored locally; no simultaneous action can modify them (via a write). Then, immediately at the end of the action, the “writes” that have been computed are written unto the global state. Since there cannot be simultaneous actions performing the same writes, no ambiguity arises.

Back to example 1, we have: $Read(add\ w\ to\ w') = \{w, w'\}$ and $Write(add\ w\ to\ w') = \{w'\}$. Consider the expression:

$$(\underbrace{add\ y\ to\ z}_a \parallel \underbrace{add\ x\ to\ y}_b; \underbrace{add\ y\ to\ z}_c) :: [\langle state, [\langle x, 2 \rangle, \langle y, 3 \rangle, \langle z, 6 \rangle] \rangle, \langle time, 0 \rangle]$$

Suppose that the maximal overhead constant is $\alpha = 1$.² A possible computation is presented in Figure 3. The actions $a = add\ y\ to\ z$ and $b = add\ x\ to\ y$ may be started in parallel, since they

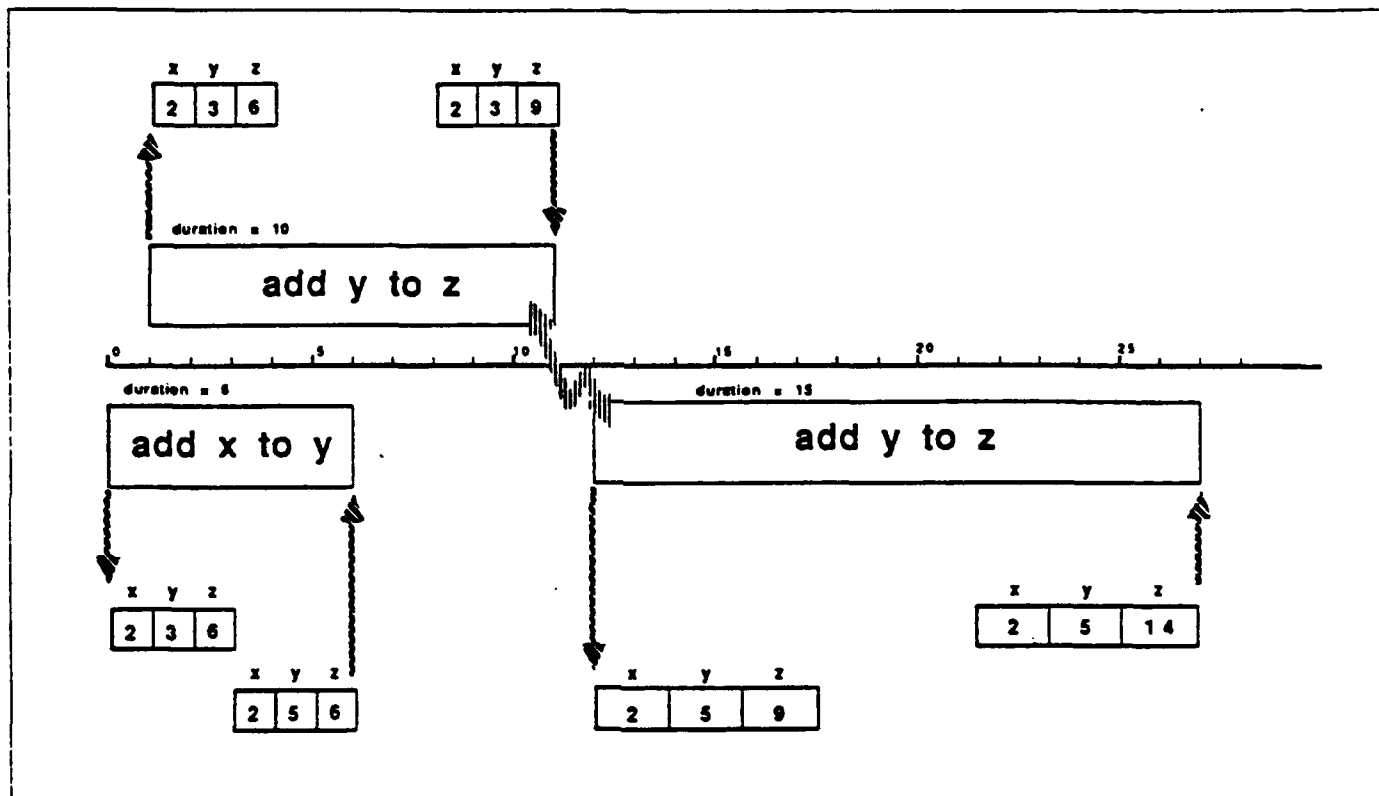


Figure 3: An Example of Computation — A Bisequential Case

modify different variables. They are preceded by respective overheads of 1 and 0, and respectively terminate at $t = 11$ and $t = 6$. Note that the value of y used throughout the action a is the initial value $y = 3$, since y is globally modified by b at $t = 6$ only. As of $t \geq 6$, the last action $c = add\ y\ to\ z$ is willing to start. However, since it produces a *variable conflict* in z with the action a that is currently running, c is automatically delayed until the end of the conflict, at $t = 11$. It then proceeds, after an overhead of 1, namely at $t = 12$. The final state is as presented in Figure 3.

²We take the unit of time equal to $10^{-3}s$.

4 Formal Semantics of Real-Time Process Specifications

4.1 Semantics for Biparallel Processes

As mentioned in the previous section, the semantics of our approach is defined a functorial transformation, that maps a *real-time process specification* \mathcal{P} into the classical, algebraic specification $\text{Sem}(\mathcal{P})$ (cf. Figure 5, page 8). Doing so, the delicate point is the treatment of the parallelism, in expressions s.a. $p :: e$, where some “ \parallel ” symbol appears in p . We firstly restrict, in this subsection, to the *biparallel* case, i.e. when $p = p_1 \parallel p_2$, where both p_1 and p_2 are the sequential composition of atomic action (actually, this is slightly more general, as shown immediately).

Define two predicates $\text{seq}, \text{biseq} : \text{process} \rightarrow \text{boolean}$ by :

$\text{seq}(a)$	
$\text{seq}(p + q) : - \text{seq}(p), \text{seq}(q)$	$\text{seq}(a; p) : - \text{seq}(p)$
$\text{biseq}(p) : - \text{seq}(p)$	$\text{biseq}(p + q) : - \text{biseq}(p), \text{biseq}(q)$
$\text{biseq}(a; p) : - \text{biseq}(p)$	$\text{biseq}(p \parallel q) : - \text{seq}(p), \text{seq}(q)$

For instance, the expression $a; (b; (c + d) \parallel e) + f; g$ is bisequential. We have in fact :

Theorem 1

A process is bisequential iff it is congruent, modulo the equations between constructors of $\text{Sem}(\mathcal{P})$ (cf. Figure 5) and modulo the equation $p; (q + r) = (p; q) + (p; r)$,³ to an expression of the kind :

$$\sum_{k=1}^K (\prod_{i=1}^{J_k} a_{i,k}) \parallel (\prod_{j=1}^{J_k} b_{j,k}) + \sum_{k=1}^{K'} \prod_{l=1}^{L_k} c_{l,k}$$

We now concentrate on defining $q :: e$, for q being biparallel. A new technical (“hidden”) operator $\begin{pmatrix} -1 \\ -2 \end{pmatrix} \parallel \begin{pmatrix} -3 \\ -4 \end{pmatrix} : \text{process} \times \text{process} \times \text{real} \times \text{real} \rightarrow \text{process}$ is introduced, so that $\begin{pmatrix} p \\ \epsilon \end{pmatrix} \parallel \begin{pmatrix} p' \\ \epsilon' \end{pmatrix}$ is like $p \parallel p'$, except that the overhead before the first action of p (resp. of p') is constrained to be equal to ϵ (resp. to ϵ') — if no conflict arises ; otherwise, the process corresponding to the smallest ϵ, ϵ' starts first after exactly $\inf\{\epsilon, \epsilon'\}$, and the second is postponed (as explained above).

The operator “ \parallel ” is then defined as follows :

$$p \parallel p' = \iint_{(0..a] \times (0..a]} \begin{pmatrix} p \\ \epsilon \end{pmatrix} \parallel \begin{pmatrix} p' \\ \epsilon' \end{pmatrix} d\epsilon d\epsilon' \quad (2)$$

We use the sign “ \int ” for a nondeterministic choice indexed over a *continuous* domain, while “ \sum ” is used for *discrete* domains (cf. also Appendix A).

The next equation (3) (presented in Figure 4) assigns semantics to the application of the last operator to an environment $[s, t]$. In (3), a, b are atomic actions, p, q are processes, and ϵ, ϵ' are in $(0..a]$. The case where p and/or q is missing is postponed to Appendix B (Equations (3)', (3)'', (3)''', Figure 7), for sake of clarity.

On the whole, the rigorous definition of $\text{Sem}(\mathcal{P})$ is summed up in Figure 5. A sufficient completeness result is as follows (the proof is in [Deutsch 91]) :

³That is *not* assumed to hold in $\text{Sem}(\mathcal{P})$, however.

$$\begin{aligned}
& \left(\begin{array}{c} a; p \\ \epsilon \end{array} \right) \parallel \left(\begin{array}{c} b; q \\ \epsilon' \end{array} \right) :: [s, t] = \text{let } \mu_a = \epsilon + \mu(a, s), \mu_b = \epsilon' + \mu(b, s) \text{ in} \\
& \quad \text{if } \mu_a \leq \epsilon' \quad \text{-- i.e. } b \text{ starts after } a \text{ is over. No conflict:} \\
& \quad \quad \text{then } (p \parallel b; q) :: [a : s, t + \mu_a] \\
& \quad \text{elseif } \mu_b \leq \epsilon \quad \text{-- i.e. } a \text{ starts after } b \text{ is over. No conflict:} \\
& \quad \quad \text{then } (a; p \parallel q) :: [b : s, t + \mu_b] \\
& \quad \text{else} \quad \text{-- } a \text{ and } b \text{ are willing to run simultaneously:} \\
& \quad \quad \text{if } \text{Write}(a) \cap \text{Write}(b) = \emptyset \quad \text{-- No conflict:} \\
& \quad \quad \quad \text{then if } \mu_a \leq \mu_b \quad \text{-- } a \text{ terminates before } b: \\
& \quad \quad \quad \quad \text{then } (p \parallel \text{wait}(\mu_b - \mu_a); q) :: [s[\text{Write}(a) \setminus a : s, \text{Write}(b) \setminus b : s], t + \mu_a] \\
& \quad \quad \quad \quad \text{elseif } \mu_b \leq \mu_a \quad \text{-- } b \text{ terminates before } a: \\
& \quad \quad \quad \quad \text{then } (\text{wait}(\mu_a - \mu_b); p \parallel q) :: [s[\text{Write}(a) \setminus a : s, \text{Write}(b) \setminus b : s], t + \mu_b] \\
& \quad \quad \quad \text{else} \quad \text{-- conflict:} \\
& \quad \quad \quad \quad \text{if } \epsilon < \epsilon' \quad \text{-- } a \text{ starts before } b, b \text{ is delayed:} \\
& \quad \quad \quad \quad \quad \text{then } (p \parallel b; q) :: [a : s, t + \mu_a] \\
& \quad \quad \quad \quad \text{elseif } \epsilon' < \epsilon \quad \text{-- } b \text{ starts before } a, a \text{ is delayed:} \\
& \quad \quad \quad \quad \quad \text{then } (a; p \parallel q) :: [b : s, t + \mu_b] \\
& \quad \quad \quad \quad \text{else} \quad \text{-- } \epsilon = \epsilon': \\
& \quad \quad \quad \quad \quad (p \parallel b; q) :: [a : s, t + \mu_a] \cup (a; p \parallel q) :: [b : s, t + \mu_b]
\end{aligned}
\tag{3}$$

Figure 4: Equation (3)

Theorem 2

For any real-time process specification \mathcal{P} , when the application of the operators “.” and “ μ ” to the atomic actions is defined in a hierarchically consistent and complete way, and likewise for the definitions of the compound actions, the associated classical specification $\text{Sem}(\mathcal{P})$ is hierarchically complete and consistent, with respect to its subspecifications and to its constructors.

In particular, it implies that given $(a_i)_{1 \leq i \leq m}, (b_j)_{1 \leq j \leq n}$: atomic actions, s : state, t : time, there exist some s_k : state, t_k, t'_k : time ($1 \leq k \leq N$) such that:

$$\text{Sem}(\mathcal{P}) = \left(\left(\prod_{i=1}^m a_i \right) \parallel \left(\prod_{j=1}^n b_j \right) \right) :: [s, t] = \bigcup_{k=1}^N [s_k, [t_k..t'_k]]$$

We denote by $\text{Mod}(\mathcal{P})$ the class of the hierarchical models ([Gaudel 85], [Bidoit 89]) of $\text{Sem}(\mathcal{P})$. A usual denote by $\mathcal{T}(\mathcal{X})^s$ the terms with variables built on the signature of $\text{Sem}(\mathcal{P})$, of sort s .

Definition 3

Two processes p and p' in $\mathcal{T}(\mathcal{X})^{\text{process}}$ are observationally equivalent, which is written $p \overset{\text{obs}}{=} p'$, if $\forall q \in \mathcal{T}(\mathcal{X})^{\text{process}}, s \in \mathcal{T}(\mathcal{X})^{\text{state}}, t \in \mathcal{T}(\mathcal{X})^{\text{time}}, (p \parallel q) :: [s, t] = (p' \parallel q) :: [s, t]$.⁴

Theorem 4

- The relation “ $\overset{\text{obs}}{=}$ ” is a congruence for the operators of $\text{Sem}(\mathcal{P})$.
- For all $p, q, r \in \mathcal{T}(\mathcal{X})^{\text{process}}$, we have : $p; (q + r) \overset{\text{obs}}{=} (p; q) + (p; r)$.

For both proofs, see [Kaplan 89] for the non real-time case, and [Deutsch 91] for the real-time case. The second property means that we have a *linear-time semantics*. This is not a surprise, since what may be observed is the final result of the application of a process to an environment; the branchings are not observable. The situation changes if we observe infinite behaviours, with traces, ready sets,

⁴The last equality sign stands for the restriction of the congruence generated by the axioms of $\text{Sem}(\mathcal{P})$ to the subspecification ENVIRONMENT.

```

spec Sem( $\mathcal{P}$ ) = hiding  $\left( \begin{smallmatrix} -1 \\ -2 \end{smallmatrix} \right) \parallel \left( \begin{smallmatrix} -3 \\ -4 \end{smallmatrix} \right)$  in
use : ENVIRONMENT
new sorts : atomic, process
generated by :
  - : atomic  $\rightarrow$  process           $a_1, \dots, a_n : \dots \rightarrow$  atomic
  - ; - : process  $\times$  process  $\rightarrow$  process  - + - : process  $\times$  process  $\rightarrow$  process
   $\left( \begin{smallmatrix} -1 \\ -2 \end{smallmatrix} \right) \parallel \left( \begin{smallmatrix} -3 \\ -4 \end{smallmatrix} \right) : \text{process} \times \text{process} \times \text{real} \times \text{real} \rightarrow \text{process}$ 
derived operators :
   $c_1, \dots, c_n : \dots \rightarrow$  process          - : - : process  $\times$  state  $\rightarrow$  state
  -  $\parallel$  - : process  $\times$  process  $\rightarrow$  process  - :: - : process  $\times$  environment  $\rightarrow$  environment
   $\mu$  - : process  $\times$  state  $\rightarrow$  time
precondition :
   $p :: [s, t]$  is-defined-when bisquential( $p$ )
equations between constructors :
   $(p + q) + r = p + (q + r)$    $p + p = p$    $(p + q); r = (p; r) + (q; r)$ 
   $p + q = q + p$    $(p; q); r = p; (q; r)$    $p + q = q + p$ 
equations defining  $c_1, \dots, c_m$  : given by the specifier
equations defining  $\parallel$  : Equation (2) (Page 6)
equations defining : : given by the specifier
equations defining  $\mu$  : given by the specifier
equations defining ::
  Equation (1), page 3
  Equation (3) (figure 4, page 7), Equations (3)', (3)'', (3)''' (figure 7, page 15)
   $a :: [s, t] = [a : s, t + \mu(a, s)]$    $p + q :: e = (p :: e) \cup (q :: e)$ 
   $p :: \emptyset = \emptyset$    $p; q :: e = q :: (p :: e)$ 
   $p :: (e \cup e') = (p :: e) \cup (p :: e')$ 
var :  $a : \text{atomic}, s : \text{state}, t : \text{time}, e, e' : \text{environment}, p, q, r : \text{process}, b_1, b_2 : \text{bool}$ 
end Sem( $\mathcal{P}$ )

```

Figure 5: Semantics of a real-time process specification

etc. (work under development).

Definition 5

- A model \mathcal{M} of $\text{Mod}(\mathcal{P})$ is observational if for any processes p, p' such that $p \overset{\text{obs}}{=} p'$, then $\mathcal{M} \models p = p'$.
- The semantics of a real-time process specification \mathcal{P} is the class $\text{Mod}^{\text{obs}}(\mathcal{P})$ of all the hierarchical, observational models of \mathcal{P} .

Several concepts developed for our previous (non real-time) process specifications [KP 87, Kaplan 89] easily extend to real-time process specifications. For instance, given \mathcal{P} and \mathcal{P}' , a process specification morphism⁵ from \mathcal{P} to \mathcal{P}' is a signature morphism $\varphi : \text{Sig}(\text{Sem}(\mathcal{P})) \rightarrow \text{Sig}(\text{Sem}(\mathcal{P}'))$, that preserves the operators " \parallel ", " $::$ ", etc.⁶ Such a φ is simply determined by how it maps the actions of \mathcal{P} into the actions of \mathcal{P}' . Classically, it induces a "forget-restrict" functor \mathcal{FR}^φ from $\text{Mod}^{\text{obs}}(\mathcal{P}')$ into $\text{Mod}(\mathcal{P})$. Then:

Definition 6

A process specification \mathcal{P} is implemented by a process specification \mathcal{P}' via a morphism φ — which

⁵Abbreviated into morphism, when no confusion arises

⁶i.e. $\varphi(\parallel_{\mathcal{P}}) = \parallel_{\mathcal{P}'}$, $\varphi(::_{\mathcal{P}}) = ::_{\mathcal{P}'}$, etc.

we write: $\mathcal{P} \xrightarrow{\varphi} \mathcal{P}'$ —, if $\mathcal{FR}^{\varphi}(\text{Mod}^{\text{obs}}(\mathcal{P}')) \subseteq \text{Mod}^{\text{obs}}(\mathcal{P})$

Intuitively, this means that properties of \mathcal{P} , applied to the “implementation” in \mathcal{P}' of entities of \mathcal{P} , should be satisfied observationally in \mathcal{P}' .

Theorem 7

The implementation relation is compositional, i.e. if $\mathcal{P} \xrightarrow{\varphi} \mathcal{P}'$ and $\mathcal{P}' \xrightarrow{\varphi'} \mathcal{P}''$, then $\mathcal{P} \xrightarrow{\varphi\varphi'} \mathcal{P}''$.

4.2 Multisequential Processes

In this subsection, we discuss how to extend the semantics of $\text{Sem}(\mathcal{P})$ to a larger class of processes. In particular, we wish to deal with *multisequential* processes, that include processes of the form $\prod_{n=1}^N \prod_{i=1}^{I_n} a_{i,n}$ (that are not bisquential if $N > 2$). As before, define *multiseq* : *process* \rightarrow *boolean* by :

$\text{multiseq}(p) : - \text{seq}(p)$	$\text{multiseq}(p + q) : - \text{multiseq}(p), \text{multiseq}(q)$
$\text{multiseq}(a; p) : - \text{multiseq}(p)$	$\text{multiseq}(p \parallel q) : - \text{seq}(p), \text{multiseq}(q)$

We have the equivalent of Theorem 1 :

Theorem 8

A process is *multisequential* iff it is congruent, modulo the equations between constructors of $\text{Sem}(\mathcal{P})$ (cf. Figure 5) and modulo the equation $p; (q + r) = (p; q) + (p; r)$, to an expression of the kind :⁷

$$\sum_{k=1}^K \parallel_{l=1}^{L_k} \prod_{i=1}^{I_{k,l}} a_{i,k,l}$$

We also let, as before :

$$\parallel_{n=1}^N p_n = \int_{(0, \infty]^N} \parallel_{n=1}^N \left(\frac{p_n}{\epsilon_n} \right) d\epsilon_1 \dots d\epsilon_N$$

(Similar to the above Equation (2)).

We concentrate on expressions such as :

$$\parallel_{n=1}^N \left(\frac{a_n; q_n}{\epsilon_n} \right) :: [s, t] \quad (4)$$

The first step is to determine which actions among the a_n 's may start first. For $n \in [1..N]$ ($= \{1, \dots, N\}$), we let $\mu_n = \epsilon_n + \mu(a_n, s)$; so that $t + \mu_n$ is the date at which a_n terminates, if it is not delayed because of conflicts. Say that two actions a_n and a_m , $n \neq m$, are in *dynamic conflict*, which we write $dc(a_n, a_m)$, if their writes intersect, and if they occur simultaneously (that is $\sup\{\epsilon_n, \epsilon_m\} \leq \inf\{\mu_n, \mu_m\}$). A subset J of $[1..N]$ is called a *clique* if it satisfies the following properties :

- (i) $\forall j, j' \text{ in } J, \neg dc(a_j, a_{j'})$ (i.e. the actions of J are pairwise conflict-free).
- (ii) Let $\bar{\epsilon} = \inf\{\epsilon_n \mid n \in [1..N]\}$. There exists some j_0 in J such that $\epsilon_{j_0} = \bar{\epsilon}$ (which means that J contains one of the actions that is ready to start first).

⁷With the the convention that $\parallel_{i=1}^1 \text{exp}_i = I$.

- (iii) Let $\bar{\mu} = \inf\{\mu_j \mid j \in J\}$. Then $\forall j \in J, \epsilon_j \leq \bar{\mu}$ (which means that each action of J should start before any other of its actions terminates).
- (iv) J is maximal with properties (i),(ii),(iii)

There may be several cliques associated to a given situation, as shown in Figure 6. All the actions of a clique will be started simultaneously — which assigns a semantics of *maximal parallelism* to our approach. Due to the maximality of J , $[1..n] - J$ may be partitioned into $\Gamma_1 \cup \Gamma_2$, where

- $\gamma \in \Gamma_1$ if a_γ is in dynamic conflict with some a_j , for $j \in J$. In this case, it is delayed until after $\bar{\mu}_\gamma = \sup\{\mu_j \mid j \in J \wedge dc(a_\gamma, a_j)\}$.
- $\gamma \in \Gamma_2$ if a_γ has no dynamic conflict with any a_j , for $j \in J$, but it starts after $\bar{\mu}$.

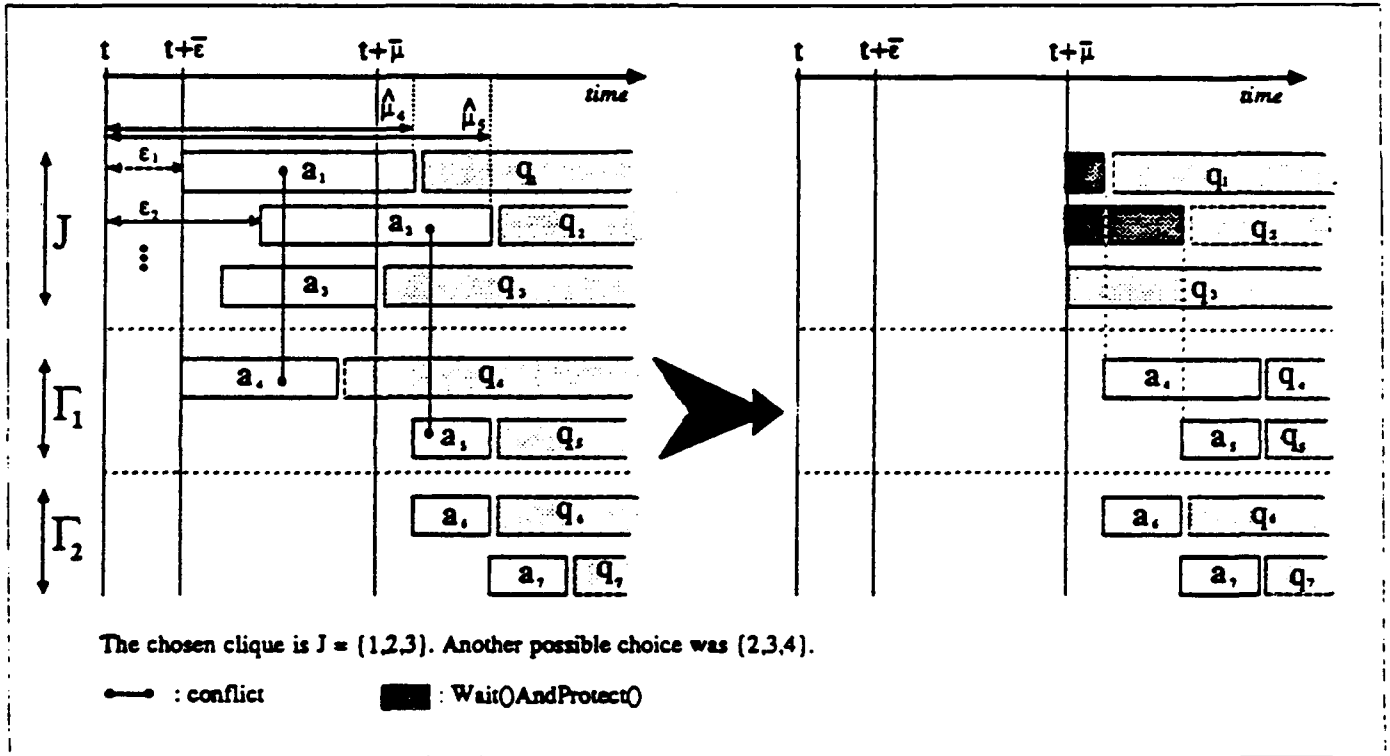


Figure 6: An Example of Computation — The Multisequential Case

Via the semantic equations, we jump to date $t + \bar{\mu}$. The new state s' is equal to s , where the writes of each a_j , $j \in J$, are replaced by $a_j : s$.⁸ In the above expression (4), each entity $\begin{pmatrix} a_n; q_n \\ \epsilon_n \end{pmatrix}$ is replaced by an expression $\begin{pmatrix} r_n \\ \zeta_n \end{pmatrix}$, as follows :

- If $n \in J$ and $\mu_n = \bar{\mu}$, then $\begin{pmatrix} r_n \\ \zeta_n \end{pmatrix} := \int_{\epsilon \in (0, \alpha)} \begin{pmatrix} q_n \\ \epsilon \end{pmatrix} d\epsilon$: since a_n has terminated, the remaining process q_n is ready to start, with any overhead smaller than α .
- If $n \in J$ and $\mu_n > \bar{\mu}$, then $r_n := \text{Wait}(\mu_n - \bar{\mu}).\text{AndProtect}(\text{Writes}(a_n)); q_n$. We use a hidden atomic action $\text{Wait}(\tau).\text{AndProtect}(X)$, that waits for a period of time τ , and during which any other action is in conflict with the variables of X .⁹ Also, $\zeta_n := 0$.

⁸Remember that these writes do not intersect.

⁹It is in fact assigned a particular semantic treatment, not detailed here.

- If $n \in \Gamma_1$, then $\begin{pmatrix} r_n \\ \zeta_n \end{pmatrix} := \int_{\epsilon \in (0, \alpha]} \begin{pmatrix} Wait(\widehat{\mu}_n - \bar{\mu}) And Protect(0); a_n; q_n \\ \epsilon \end{pmatrix} d\epsilon.$
- If $n \in \Gamma_2$, then $r_n = a_n; q_n$ and $\zeta_n = \epsilon_n - \bar{\mu}$

The reader is urged to follow the concepts introduced in these last lines on Figure 6. Finally, one has the (semi-formal) semantic equation :

$$\left\| \begin{pmatrix} a_n; q_n \\ \epsilon_n \end{pmatrix} \right\|_{n=1}^N :: [s, t] = \sum_{J : \text{clique of } [1..N]} \left\| \begin{pmatrix} r_n \\ \epsilon_n \end{pmatrix} \right\|_{n=1}^N :: [s', t + \bar{\mu}] \quad (5)$$

with s' as above.

Note : if some a_i has an associated continuation p_i , it will eventually vanish from the expression p under consideration — the degree of parallelism of p being decreased by one. This implies an easy modification of the above Equation (5) — similar to the modification of Equation (3) of $\text{Sem}(\mathcal{P})$ into Equations (3')(3'')(3''') (cf. Appendix B and Figure 7).

4.3 The General Case

So far, semantics for $p :: [s, t]$ has been defined for p being bisquential, and then multisequential, which still excludes processes s.a. $p = ((a; b \parallel c); d) \parallel (e; f)$.¹⁰ In this subsection, we explain *informally* how this is done; technical details are worked out in [Deutsch 91]. A function $\text{Firsts} : \text{process} \rightarrow \text{set-of-atomic}$ is defined, which determines the set of atomic actions that are ready to start. With p as above, $\text{Firsts}(p) = \{a, c, e\}$. We then proceed as above : each action a_i among the *firsts* of p is tentatively assigned a delay $\epsilon_i \in (0, \alpha]$. The possible cliques J are computed, with the associated sets Γ_1 and Γ_2 . And finally, each action is *substituted* inside p according to the rules given above. For instance, suppose that $J = \{a, e\}$, that $\bar{\mu}$ is reached for a , and that c is in dynamic conflict with e (and $\widehat{\mu}_c = \mu_e$). The semantic equations will allow to jump to date $t + \bar{\mu}$, with a state modified by the application of a and e , and the current process being $(b \parallel Wait(\widehat{\mu}_c - \bar{\mu}).AndProtect(0)); d \parallel Wait(\mu_e - \bar{\mu}).AndProtect(Writes(e)); f$.

5 Extensions to the Basic Formalism

5.1 Wait Actions

The specifier is provided with two possible *wait* actions:

- $wait(\zeta)$, that consists in waiting for a fixed amount of time ζ . It is simply an atomic action, characterized by the relations $wait(\zeta) : s = s$ and $\mu(wait(\zeta), s) = \zeta$.
- $wait-until(b)$, where b is a boolean expression. Such a b may be applied to an environment $[s, t]$ via a special-purpose operator $\overset{bool}{:}$, yielding *true* or *false*. For instance, if $b = (\text{time} > 3)$, then

¹⁰Note that such processes would not be well-defined — in fact, all processes would be multisequential — if we had assigned to operator “ \parallel ” the profile $\text{action} \times \text{process} \rightarrow \text{process}$, as in CCS, CSP or several versions of ACP

$b^{bool} : [< \text{state}, s >, < \text{time}, 3.2 >] = \text{true}$.¹¹ Now, semantics of *wait-until* is expressed by the following equation :

$$\begin{aligned} \text{wait-until}(b) :: [s, t] &= \text{if } b^{bool} : [s, t] = \text{true} \\ &\quad \text{then } [s, t] \\ &\quad \text{else } \bigcup_{\epsilon \in (0, \alpha]} (\text{wait-until}(b) :: [s, t + \epsilon]) \end{aligned} \quad (6)$$

Note that, in order to simplify the matter, we do not take into account the time needed to evaluate an expression $b^{bool} : [s, t]$. Also, the equations defining the application of parallelism have to be extended. The same two remarks apply to guarded commands, hereafter. Again, we refer to [Deutsch 91] for full details.

5.2 Guarded Commands

The specifier may refer to *guarded commands*, of the form $b_1 \rightarrow p_1 \parallel b_2 \rightarrow p_2$, where b_1 and b_2 are boolean expressions (as above). Semantics is defined by :

$$\begin{aligned} b_1 \rightarrow p_1 \parallel b_2 \rightarrow p_2 :: e &= \text{if } (b_1^{bool} : e) = \text{true} \wedge (b_2^{bool} : e) = \text{true} && \text{then } (p_1 :: e) \cup (p_2 :: e) \\ &\quad \text{elseif } (b_1^{bool} : e) = \text{true} && \text{then } p_1 :: e \\ &\quad \text{elseif } (b_2^{bool} : e) = \text{true} && \text{then } p_2 :: e \\ &\quad \text{else } \emptyset \end{aligned}$$

6 Discussion

In this article, we have provided semantics for our real-time process specifications. The mathematical, algebraic treatment of such entities turns out to be fairly complicated. Moreover, we have presented here a simplified version of a complete semantical treatment (given in [Deutsch 91]). The mathematical complexity stems from the fact that an important number of delicate issues are taken into account : conflicts, system overhead, etc. Recently, [BB 89] has presented semantics for real-time *process algebra*. Their approach is more simple than ours, but addresses less delicacies.

In [Deutsch 91], we present a primitive allowing to put together, in parallel, real-time process specifications (and not only processes of one given specification), in the flavour of [Kaplan 89]. In this case, each process specification has its independent notion of time. Inter-specification communication is by shared data, modeled by arbitrarily complex data or process specifications. In particular, this allows to simulate communication by *channels* : each channel becomes an (instance of a) process specification. It becomes easy to describe different classes of channels (reliable, unreliable, etc.) in a generic fashion.

References

- [BB 89] J. Baeten, J. Bergstra, *Real time process algebra*, CWI Report P8916, Amsterdam (1989)

¹¹ Boolean expressions and the operator b^{bool} are defined rigorously in [Kaplan 88].

- [BK 84] J. Bergstra, J. Klop, *Algebra of communicating processes with abstraction*, CWI Report CS-R8403, Amsterdam (1984)
- [Bidoit 89] M. Bidoit, *Pluss, un langage pour le développement de spécifications modulaires*, Thèse d'Etat, University of Paris-Sud.
- [Boudol 84] G. Boudol, *An asynchronous calculus MEIJE*, in NATO summer school, La-Colle-sur-Loup, France (1984).
- [Da 85] B. Desarathy, *Timing constraints of real-time systems : Constructs for expressing them and methods for validating them*, IEEE Trans. on Softw. Eng., SE-11, pp. 80-86 (1985)
- [Deutsch 91] G. Deutsch, *Abstraction and real-time issues in process specifications*, Ph.D., Bar-Ilan University, Israel (to appear 1990).
- [Gaudel 85] M.-C. Gaudel, *Towards structured algebraic specifications*, Esprit Technical Week, Bruxelles, Springer Verlag (1985).
- [Hoare 78] C.A.R. Hoare, *Communication sequential Processes*, C.A.C.M. 21, pp. 666-677 (1978).
- [JM 86] F. Jahanian, A.K. Mok, *A graph-theoretic approach for timing analysis in real-time logic*, IEEE Real-Time Systems Symposium, pp. 98-108 (1986)
- [KP 87] S. Kaplan, A. Pnueli, *Specification and implementation of concurrently accessed data structures*, Proc. of the 4th STACS Conf., LNCS 247, Springer-Verlag (1987).
- [Kaplan 88] S. Kaplan, *Spécification algébrique de types de données à accès concurrent*, Thèse d'Etat, University of Paris-Sud (1987).
- [Kaplan 89] S. Kaplan, *Algebraic specifications of concurrent systems*, in *Theoretical Computer Science* (1989).
- [KSR 88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, S. Arun-Kumar, *Compositional semantics for real-time distributed computing*, Information and Computation 79, pp. 210-256 (1988)
- [LL 83] G. Le Lann, *On real-time distributed computing*, Proc. IFIP-83 (1983).
- [Milner 80] R. Milner, *A calculus of communicating processes*, LNCS 92, Springer-Verlag (1980).
- [Wirsing 91] M. Wirsing, *Algebraic specifications*, in *Handbook of Theoretical Computer Science*, Van Leuwen Ed., Academic Press (to appear 1991).
- [Wirth 77] N. Wirth, *Towards a discipline of real-time programming*, CACM 20, pp. 577-583 (1977)

A Appendix

For some given *real-time process specification* \mathcal{P} , the models of $\text{Mod}(\mathcal{P})$ are classical, hierarchical models, except about the way the sort *time* is defined, and the way the “ \cup ” and the “ $+$ ” symbols should be understood. These two point are discussed in the present appendix.

A.1 The Notion of Time

All over this article, we supposed that time is isomorphic to the real half-line $[0.. + \infty)$. This means that are taken into account, in $\text{Mod}(\mathcal{P})$, *only* the models M such that the carrier set M^{time} of M interpreting the sort *time* is constrained to be isomorphic to $[0.. + \infty)$. Likewise, operators of sort *time* (s.a. " \leq ", " $+$ ", etc.) receive their standard interpretations over the real numbers.

Other notions of time are possible. For instance, one may decide that events may occur only at multiples of the clock rate τ . It means that we now restrict to models for which the interpretation of the sort *time* is $\tau\mathcal{N} = \{0, \tau, 2\tau, \dots\}$. Very little has to be modified w.r.t. what has been described so far. The maximal overhead α has to be rounded up unto $\lfloor \alpha/\tau \rfloor \tau$. An equation defining the duration of an atomic action, of the form $\mu(a, s) = \text{expr}$, has to be transformed into $\mu_\tau(a, s) = \lceil \text{expr}/\tau \rceil \tau$. Likewise, Equation (2), page 6 becomes :

$$(p \parallel p') = \sum_{0 < n, n' \leq \lfloor \alpha/\tau \rfloor} \binom{p}{n\tau} \parallel \binom{p'}{n'\tau} \quad (7)$$

Prototyping aspects (that is, how to symbolically execute computation in $\text{Sem}(\mathcal{P})$) become much simpler, since all the objects under consideration are in finite numbers.

Lastly, more general notion of time may be considered, expressing for instance causality relation. Time would be then given p.o. set. This point is currently under investigation.

A.2 Interpretation of the Union and of the Nondeterministic Choice

We restrict attention to models for which the interpretation of the operators $\cup : \text{environment} \times \text{environment} \rightarrow \text{environment}$ and $\emptyset : \rightarrow \text{environment}$ receive their usual *set-theoretic* interpretation.

B Appendix

Equation (3), that defines $\binom{a; p}{\epsilon} \parallel \binom{b; q}{\epsilon'}$ has to be modified into equations (3)', (3)", (3)''', as follows, in order to take count of the cases where p and/or q are missing (cf. Figure 7).

```

 $\begin{pmatrix} a \\ \epsilon \end{pmatrix} \parallel \begin{pmatrix} b; q \\ \epsilon' \end{pmatrix} :: [s, t] = \text{let } \mu_a = \epsilon + \mu(a, s), \mu_b = \epsilon' + \mu(b, s) \text{ in}$ 
  if  $\mu_a \leq \epsilon'$  -- i.e. b starts after a is over. No conflict:
    then  $b; q :: [a : s, t + \mu_a]$ 
  elseif  $\mu_b \leq \epsilon$  -- i.e. a starts after b is over. No conflict:
    then  $(a \parallel q) :: [b : s, t + \mu_b]$ 
  else -- a and b are willing to run simultaneously:
    if  $Write(a) \cap Write(b) = \emptyset$  -- No conflict:
      then if  $\mu_b \leq \mu_a$  -- a terminates before b:
        then  $wait(\mu_b - \mu_a); q :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_a]$ 
        elseif  $\mu_b \leq \mu_a$  -- b terminates before a:
          then  $(wait(\mu_a - \mu_b) \parallel q) :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_b]$ 
      else -- conflict:
        if  $\epsilon < \epsilon'$  -- a starts before b. b is delayed:
          then  $b; q :: [a : s, t + \mu_a]$ 
        elseif  $\epsilon' < \epsilon$  -- b starts before a. a is delayed:
          then  $(a \parallel q) :: [b : s, t + \mu_b]$ 
        else  $b; q :: [a : s, t + \mu_a] \cup (a \parallel q) :: [b : s, t + \mu_b]$ 

```

(3)'

```


$$\left( \begin{smallmatrix} a; p \\ \epsilon \end{smallmatrix} \right) \parallel \left( \begin{smallmatrix} b \\ \epsilon' \end{smallmatrix} \right) :: [s, t] = \text{let } \mu_a = \epsilon + \mu(a, s), \mu_b = \epsilon' + \mu(b, s) \text{ in}$$

  if  $\mu_a \leq \epsilon'$  -- i.e. b starts after a is over. No conflict:
    then  $(p \parallel b) :: [a : s, t + \mu_a]$ 
  elseif  $\mu_b \leq \epsilon$  -- i.e. a starts after b is over. No conflict:
    then  $a; p :: [b : s, t + \mu_b]$ 
  else -- a and b are willing to run simultaneously:
    if  $Write(a) \cap Write(b) = \emptyset$  -- No conflict:
      then if  $\mu_a \leq \mu_b$  -- a terminates before b:
        then  $(p \parallel wait(\mu_b - \mu_a)) :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_a]$ 
        elseif  $\mu_b \leq \mu_a$  -- b terminates before a:
          then  $wait(\mu_a - \mu_b); p :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_b]$ 
      else -- conflict:
        if  $\epsilon < \epsilon'$  -- a starts before b. b is delayed:
          then  $(p \parallel b) :: [a : s, t + \mu_a]$ 
        elseif  $\epsilon' < \epsilon$  -- b starts before a. a is delayed:
          then  $a; p :: [b : s, t + \mu_b]$ 
        else  $(p \parallel b) :: [a : s, t + \mu_a] \cup a; p :: [b : s, t + \mu_b]$ 

```

(3)''

```

 $\begin{pmatrix} a \\ c \end{pmatrix} \parallel \begin{pmatrix} b \\ c' \end{pmatrix} :: [s, t] = \text{let } \mu_a = c + \mu(a, s), \mu_b = c' + \mu(b, s) \text{ in}$ 
  if  $\mu_a \leq c'$  -- i.e. b starts after a is over. No conflict:
    then  $b :: [a : s, t + \mu_a]$ 
  elseif  $\mu_b \leq c$  -- i.e. a starts after b is over. No conflict:
    then  $a :: [b : s, t + \mu_b]$ 
  else -- a and b are willing to run simultaneously:
    if  $Write(a) \cap Write(b) = \emptyset$  -- No conflict:
      then if  $\mu_a \leq \mu_b$  -- a terminates before b:
        then  $wait(\mu_b - \mu_a) :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_a]$ 
        elseif  $\mu_b \leq \mu_a$  -- b terminates before a:
          then  $wait(\mu_a - \mu_b) :: [s[Write(a) \setminus a : s, Write(b) \setminus b : s], t + \mu_b]$ 
      else -- conflict:
        if  $c < c'$  -- a starts before b. b is delayed:
          then  $b :: [a : s, t + \mu_a]$ 
        elseif  $c' < c$  -- b starts before a. a is delayed:
          then  $a :: [b : s, t + \mu_b]$ 
        else  $b :: [a : s, t + \mu_a] \cup a :: [b : s, t + \mu_b]$ 

```

(3)'''

Figure 7: Equations (3)', (3)'', (3)'''

STRUCTURE OF CONCURRENCY

Ryszard Janicki

Department of Computer Science and Systems

McMaster University

Hamilton, Ontario, Canada, L8S 4K1

Maciej Koutny

Computing Laboratory

The University

Newcastle upon Tyne NE1 7RU, U.K.

'True concurrency' semantics assume that behavioural properties of systems can be adequately modelled in terms of causal partial orders (see [Pr86]). This assumption is arbitrary and the model, although very successful in general, seems to be insufficient to describe properly some aspects of systems with priorities, inhibitor Petri nets and error recovery systems (see [Ja87], [La85], [JK90], [JK91]). We claim that the structure of concurrency is richer, with causality being only one of several invariants generated by a set of equivalent executions or observations. The model we are going to present is a three-level model: Systems-Invariants-Observations, and we will proceed from the bottom to the top of hierarchy. Here we shall concentrate on the invariant level.

OBSERVATIONS We define observation as an abstract model of an execution of a concurrent system. More precisely, by an observation we will mean a special report supplied by an observer who has to fill in a (possibly infinite) matrix with rows and columns indexed by event occurrences. The observer is supposed to fill in the entire matrix except the diagonal, and can use only the symbols: \rightarrow , \leftarrow and \leftrightarrow ; where \rightarrow denotes precedence, \leftarrow following and \leftrightarrow simultaneity. Together with a natural interpretation of the precedence relation, this means that we consider observations which can be represented by *partially ordered sets* of event occurrences, where ordering represents *precedence*, and incomparability represents *simultaneity*. A *partially ordered set* (or *poset*) is a pair $po = (X, R)$, where X is a non-empty set and $R \subseteq X \times X$ is an irreflexive and transitive relation. po is *initially finite* if for every $a \in X$, the set $\{b \in X \mid \neg aRb\}$ is finite. po is an *interval order* [Fi70] if for all $a, b, c, d \in X$, $(aRb \wedge cRd) \Rightarrow (aRd \vee cRb)$. We also denote: $\rightarrow_{po} = R$, $\leftarrow_{po} = R^{-1}$ and $\leftrightarrow_{po} = \{(a, b) \in X \times X \mid a \neq b \wedge \neg aRb \wedge \neg bRa\}$.

Not all partial orders may be interpreted as valid observations. The additional conditions we require are that the observer perceives only a single thread of time and can only observe a finite number of events in a finite period of time. Moreover, an event can last only for a finite period of time. A formal analysis of these assumptions leads to the following definition (see [JK90]).

Observation is an initially finite interval order of event occurrences.

INVARIANTS AND HISTORIES Describing a concurrent system solely in terms of the observations it may generate is unsatisfactory for many reasons. In fact any argument made in favour of causality (see, e.g., [BD85]), can also support the introduction of the new invariants. To define such invariants we now will focus on the relationship between observations of the same concurrent history, where a concurrent history is essentially an invariant or a *set of invariants* satisfied by all its observations.

A *report set* is a non-empty set Δ of observations with the same domain $dom(\Delta)$. A report set may be considered as the first approximation of a concurrent history. A *simple (binary) relational invariant* of Δ , $I \in SRI(\Delta)$, is a binary relation on $dom(\Delta)$ which can be characterised by $(a,b) \in I \Leftrightarrow a \neq b \wedge \forall o \in \Delta. \Phi(a,b,o)$, where $\Phi(a,b,o)$ is a formula defined by the grammar: $\Phi := true \mid false \mid a \rightarrow_o b \mid a \leftarrow_o b \mid a \leftrightarrow_o b \mid \neg \Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi$. Let $\rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightrightarrows_\Delta, \nearrow_\Delta$ and \nwarrow_Δ be binary relations on $dom(\Delta)$ defined in the following way.

$$\begin{aligned} a \rightarrow_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \rightarrow_o b & a \leftarrow_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \leftarrow_o b \\ a \leftrightarrow_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \leftrightarrow_o b & a \rightrightarrows_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \rightarrow_o b \vee a \leftarrow_o b \\ a \nearrow_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \rightarrow_o b \vee a \leftrightarrow_o b & a \nwarrow_\Delta b &: \Leftrightarrow a \neq b \wedge \forall o \in \Delta. a \leftarrow_o b \vee a \leftrightarrow_o b. \end{aligned}$$

The relations $\rightarrow_\Delta, \leftarrow_\Delta$ are called *causalities*, \rightrightarrows_Δ *commutativity*, \leftrightarrow_Δ *synchronisation*, and $\nearrow_\Delta, \nwarrow_\Delta$ *weak causalities*. We will use $\rightarrow, \leftarrow, \leftrightarrow, \rightrightarrows, \nearrow$ and \nwarrow to denote mappings which for every report set return respectively $\rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightrightarrows_\Delta, \nearrow_\Delta$ and \nwarrow_Δ . We call these mappings *invariants*, *SRI*.

THEOREM 1 $SRI(\Delta) = \{\emptyset, \rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightrightarrows_\Delta, \nearrow_\Delta, \nwarrow_\Delta, dom(\Delta) \times dom(\Delta) - id_{dom(\Delta)}\}$, and there is Δ such that $SRI(\Delta)$ consists of eight different relations. \square

Due to the symmetry present in $SRI(\Delta)$ we can in fact consider only four non-trivial invariants, namely: $\rightarrow_\Delta, \leftrightarrow_\Delta, \rightrightarrows_\Delta, \nearrow_\Delta$. Furthermore, \rightarrow_Δ and \leftarrow_Δ may be expressed in terms of \nearrow_Δ and \rightrightarrows_Δ , so it seems reasonable to ask how to find a possibly smallest sets of invariants from which all the relations in $SRI(\Delta)$ could be generated.

A *signature* of a non-empty family F of report sets is a set of invariants $S \subseteq SRI$ such that for all $\Delta, \Delta_o \in F$, $(dom(\Delta) = dom(\Delta_o) \wedge \forall I \in S. I(\Delta) = I(\Delta_o)) \Rightarrow (\forall I \in SRI. I(\Delta) = I(\Delta_o))$. S is *universal* if F is the family of all report sets. S is *minimal* if no proper subset of S is a signature of F and, furthermore, for all $J \in S$ and $I \in SRI - S$, if $I(\Delta) \subseteq J(\Delta)$ for all report sets Δ then $(S - \{J\}) \cup \{I\}$ is not a signature of F . It turns out that $\{\nearrow, \rightrightarrows\}$ and $\{\nwarrow, \rightrightarrows\}$ are the only universal minimal signatures.

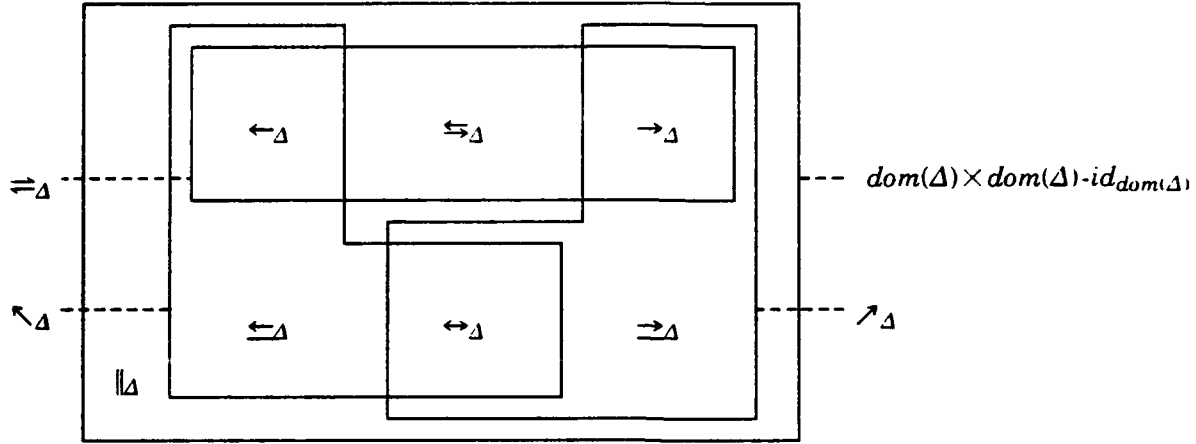
A *history* is a report set Δ which is a complete representation of some phenomena underlying the reports of Δ . This is to be captured by requiring that Δ includes all the reports satisfying the relevant structural properties. In our approach these properties are expressed using the simple report invariants.

For every $I \in SRI$, let Φ_I denote any formula (see Theorem 1) such that $(a,b) \in I(\Delta) \Leftrightarrow \forall o \in \Delta. \Phi_I(a,b,o)$. Let Δ be a report set and $S \subseteq SRI$. The *S-closure* of Δ , denoted $\Delta^{(S)}$, is the set comprising all observations o with the domain $dom(\Delta)$ such that for all $I \in S$, $(a,b) \in I(\Delta) \Rightarrow \Phi_I(a,b,o)$. It turns out that $\Delta \subseteq \Delta^{(S)}$, and if S is a universal signature then $\Delta^{(S)} = \Delta^{(SRI)}$. We now can introduce the central notion of this paper by saying that a history is a report set which can be unambiguously and fully described by the invariants it generates.

A history, $\Delta \in Hist$, is a non-empty report set Δ such that $\Delta = \Delta^{(SRI)}$.

COMPONENTS AND PARADIGMS For a concurrent history Δ the set $SRI(\Delta)$ can be treated just as any finite family of sets. In particular, one can look at the components $CSRI(\Delta) = \{\rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightrightarrows_\Delta, \nearrow_\Delta, \nwarrow_\Delta, id_\Delta\}$ defined by this family as shown in the diagram below.

A formula which says that a given relationship between two event occurrences a and b has been observed in Δ is called a *simple trait*. We have three kinds of simple traits: $\psi_{\rightarrow} \equiv \exists o \in \Delta. a \rightarrow_o b$,



$\psi_{\leftarrow} \equiv \exists o \in \Delta. a \leftarrow_o b$ and $\psi_{\leftrightarrow} \equiv \exists o \in \Delta. a \leftrightarrow_o b$. Clearly, the components in $CSRI(\Delta)$ can be defined using the simple traits, e.g., $a ||_{\Delta} b \Leftrightarrow \psi_{\rightarrow} \wedge \psi_{\leftarrow} \wedge \psi_{\leftrightarrow}$ and $a \rightarrow_{\Delta} b \Leftrightarrow \psi_{\rightarrow} \wedge \neg \psi_{\leftarrow} \wedge \neg \psi_{\leftrightarrow}$. Since $\rightarrow_{\Delta} = (\leftarrow_{\Delta})^{-1}$ and $\rightrightarrows_{\Delta} = (\leftrightsquigarrow_{\Delta})^{-1}$ we only need to discuss five components: \rightarrow_{Δ} , $||_{\Delta}$, $\leftrightsquigarrow_{\Delta}$, \leftrightarrow_{Δ} and $\rightrightarrows_{\Delta}$. The first component (and an invariant) is *causality*. $||_{\Delta}$ should be interpreted as *concurrency* (two events can be observed simultaneously and in both orders). Both causality and concurrency can be found in the 'true concurrency' models. $\leftrightsquigarrow_{\Delta}$ represents what is usually referred to as *interleaving* (two events can be observed in both orders, but not simultaneously), and is usually dealt with on the level of observations rather than invariants. The fourth component (and an invariant) can be interpreted as *synchronisation*. It is currently introduced only in its implicit form, e.g. as a silent action in CCS [Mi80]. $\rightrightarrows_{\Delta}$ is not to our knowledge a part of any of the existing models. We think it captures *disabling* of an event by another event (see [Ja87]).

The approach to concurrency which is based entirely on the concept of causality relation requires that for every concurrent history: if two event occurrences have been observed simultaneously, then it is possible to observe them in both orders, and vice versa. This means that every concurrent history besides being invariant-closed, must also adhere to the following rule: $(\exists o \in \Delta. a \leftrightarrow_o b) \Leftrightarrow ((\exists o \in \Delta. a \rightarrow_o b) \wedge (\exists o \in \Delta. a \leftarrow_o b))$. We will call such rules *paradigms*, as they characterise the general internal structure of concurrent histories.

The *paradigms*, $\omega \in Par$, are defined by the grammar: $\omega := true \mid false \mid \psi_{\rightarrow} \mid \psi_{\leftarrow} \mid \psi_{\leftrightarrow} \mid \neg \omega \mid \omega \vee \omega \mid \omega \wedge \omega \mid \omega \Rightarrow \omega$. The evaluation of the formulas $\omega \in Par$ follows the standard rules. A history $\Delta \in Hist$ satisfies a paradigm $\omega \in Par$ if for all $a, b \in dom(\Delta)$, $a \neq b \Rightarrow \omega(a, b, \Delta)$. We denote this by $\Delta \in Par(\omega)$. Two paradigms, ω and ω_0 , are *equivalent*, $\omega \sim \omega_0$, if $Par(\omega) = Par(\omega_0)$. Some of the paradigms are equivalent, reducing the number of cases we have to consider. Let $\omega_1 \equiv \psi_{\leftrightarrow} \Rightarrow \psi_{\rightarrow} \vee \psi_{\leftarrow}$, $\omega_2 \equiv \psi_{\rightarrow} \wedge \psi_{\leftarrow} \Rightarrow \psi_{\leftrightarrow}$, $\omega_3 \equiv \psi_{\rightarrow} \wedge \psi_{\leftrightarrow} \Rightarrow \psi_{\leftarrow}$, $\omega_4 \equiv \psi_{\rightarrow} \Rightarrow \psi_{\leftarrow} \vee \psi_{\leftrightarrow}$ and $\omega_5 \equiv \psi_{\rightarrow} \wedge \psi_{\leftarrow} \wedge \psi_{\leftrightarrow} \Rightarrow false$. It turns out that (up to $\neg Par = \{\omega_{i_1} \wedge \dots \wedge \omega_{i_k} \mid k \leq 5 \wedge i_j \leq 5\}$) which means that there are $2^5 = 32$ possible paradigms for the report system of concurrent observations. However, the nature of problems considered in computer science implies that two of the ω_i 's may be safely rejected. The first 'unreasonable' ω_i is ω_4 which excludes the sequential composition construct. Similarly, we reject ω_5 since it

excludes the standard parallel composition operator. Hence we have $2^3=8$ paradigms to consider:
 $\pi_1 \equiv \text{true}$, $\pi_2 \equiv \omega_1$, $\pi_3 \equiv \omega_2$, $\pi_4 \equiv \omega_3$, $\pi_5 \equiv \omega_1 \wedge \omega_2$, $\pi_6 \equiv \omega_1 \wedge \omega_3$, $\pi_7 \equiv \omega_2 \wedge \omega_3$ and $\pi_8 \equiv \omega_1 \wedge \omega_2 \wedge \omega_3$

THEOREM 2 (relationship between the components and paradigms) Let $\Delta \in \text{Hist}$.

- | | |
|--|---|
| 1. $\Delta \in \text{Par}(\pi_1)$. | 5. $\Delta \in \text{Par}(\pi_5) \Leftrightarrow \leftrightarrow_\Delta = \Rightarrow_\Delta = \emptyset$. |
| 2. $\Delta \in \text{Par}(\pi_2) \Leftrightarrow \leftrightarrow_\Delta = \emptyset$. | 6. $\Delta \in \text{Par}(\pi_6) \Leftrightarrow \leftrightarrow_\Delta = \Rightarrow_\Delta = \emptyset$. |
| 3. $\Delta \in \text{Par}(\pi_3) \Leftrightarrow \Leftarrow_\Delta = \emptyset$. | 7. $\Delta \in \text{Par}(\pi_7) \Leftrightarrow \Leftarrow_\Delta = \Rightarrow_\Delta = \emptyset$. |
| 4. $\Delta \in \text{Par}(\pi_4) \Leftrightarrow \Rightarrow_\Delta = \emptyset$. | 8. $\Delta \in \text{Par}(\pi_8) \Leftrightarrow \leftrightarrow_\Delta = \Leftarrow_\Delta = \Rightarrow_\Delta = \emptyset$. \square |

Paradigm π_1 simply admits all concurrent histories. The most restrictive paradigm, π_8 , admits concurrent histories Δ such that $\exists o \in \Delta. a \leftrightarrow_o b \Leftrightarrow (\exists o \in \Delta. a \rightarrow_o b) \wedge (\exists o \in \Delta. b \rightarrow_o a)$. It is the paradigm adopted by the existing models supporting 'true concurrency' semantics.

THEOREM 3 $\{\nearrow, \Leftarrow\}$ is a minimal signature for $\text{Par}(\pi_1)$ and $\text{Par}(\pi_2)$. $\{\Leftarrow, \rightarrow\}$ is a minimal signature for $\text{Par}(\pi_4)$ and $\text{Par}(\pi_6)$. $\{\rightarrow, \nearrow\}$ is a minimal signature for $\text{Par}(\pi_3)$ and $\text{Par}(\pi_5)$. Moreover, $\{\nearrow\}$ is a minimal signature for $\text{Par}(\pi_7)$ and $\{\rightarrow\}$ is a minimal signature for $\text{Par}(\pi_8)$. \square

Thus when the law $\exists o \in \Delta. a \leftrightarrow_o b \Leftrightarrow (\exists o \in \Delta. a \rightarrow_o b) \wedge (\exists o \in \Delta. b \rightarrow_o a)$ holds, then \rightarrow_Δ , i.e. causality, is *the only invariant that we need*, and this fact is a *theorem* in our approach. We note that in the most general case (i.e. $\text{Par}(\pi_1)$) the explicit causality invariant is not needed.

SYSTEMS In [JK91] we described how the new invariants might be used to define a 'truly concurrent' semantics of inhibitor Petri nets under the assumption that all observations are step sequences, and that every history considered belongs to π_3 . (A step sequence is an initially finite poset po such that $(a \leftrightarrow_{po} b \wedge b \leftrightarrow_{po} c \wedge a \neq c) \Rightarrow a \leftrightarrow_{po} c$.)

ACKNOWLEDGEMENT This research was partially supported by a grant from NSERC, No. OGP 0036539 and by ESPRIT Basic Research Action 3148 (project DEMON).

REFERENCES

- [BD85] Best E., Devillers R., *Concurrent Behaviour: Sequences, Processes and Programming Languages*, GMD-Studien Nr. 99, GMD, Bonn, 1985.
- [Fi70] Fishburn P.C., *Intransitive Indifference with Unequal Indifference Intervals*, J. Math. Psych. 7, 1970, pp. 144-149.
- [Ja87] Janicki R., *A Formal Semantics for Concurrent Systems with a Priority Relation*, Acta Informatica 24, 1987, pp.33-55.
- [JK90] Janicki R., Koutny M., *A Bottom-Top Approach to Concurrency Theory Part I: Observations, Invariants and Paradigms*, TR 90-4, McMaster University, Hamilton.
- [JK91] Janicki R., Koutny M., *Invariants and Paradigms of Concurrency Theory*, Proc. of PARLE'91, Lecture Notes in Computer Science, to appear.
- [La85] Lamport L., *What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority*, 12th ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, 1985, pp. 78-83.
- [Mi80] Milner R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer 1980.
- [Pr86] Pratt V., *Modelling Concurrency with Partial Orders*, Int. Journal of Parallel Programming 15, 1 (1986), pp. 33-71.

On Rewriting Behavioural Semantics in Process Algebras

P. Inverardi ♥, M. Nesi♥♦

♥ I.E.I.- C.N.R., via S. Maria 46, I-56126 Pisa, ITALY

♦ Computer Laboratory, University of Cambridge, Cambridge, UK

Introduction

One of the most interesting features of specification languages for concurrent systems like CCS, CSP, ACP, LOTOS, etc., is their algebraic nature. This allows for an algebraic characterization of their semantics, besides the usual operational one that is based on the labelled transition systems interpretation of the language. As it is well known, it is possible to equip these languages with several different semantics that define which processes can be considered as equivalent with respect to a certain *behaviour*. In the past few years there has been a growing interest in the field of the analysis and verification of properties for these languages and a number of tools and approaches have been proposed and realized [Pr89, Pr90].

In this framework, we have undertaken a project whose main goal is to develop a verification system for process algebras formalisms entirely based on equational reasoning. That is, our main concern is in providing tools and methods to reason about concurrent systems by exploiting the algebraic nature of the language in which the systems are specified. There are several motivations to this choice both of methodological and practical nature. It is out of the scope of this paper to discuss them and we refer to [DIN90, CIN91] for a detailed motivation of the approach.

Here we present the problems we have addressed so far and discuss the solutions we have adopted. To be more precise, we analyse one aspect of the kind of reasoning we want to do, the so-called automatic reasoning, that is, the possibility of equipping the axiomatic presentations of various behavioural equivalences with equivalent rewriting relations. As it will be clear in the following, this is not straightforward even in the case of finite processes. The interesting point is that, some of these equivalent rewriting relations are actually *strategies*, since they are driven by a control structure which properly decides when (and where) a rewriting step has to be applied.

Setting the framework

We consider the following subset of CCS [Mil89] whose syntax is:

$$E ::= \text{NIL} \mid \mu.E \mid E+E \mid X \mid \text{rec}X.E$$

L is the set of observable actions ranged over by λ , $\tau \notin L$ is the unobservable action and $L \cup \{\tau\}$ is the set of basic actions ranged over by μ . The operational semantics is given by the following inference rules:

$$\text{act.} \quad \mu.E \rightarrow \mu.E$$

$$\text{sum.} \quad E_1 \rightarrow E_2 \text{ implies } E_1 + E \rightarrow E_2 \text{ and } E + E_1 \rightarrow E_2$$

$$\text{rec.} \quad E(\text{rec}X.E/X) \rightarrow E_1 \text{ implies } \text{rec}X.E \rightarrow E_1$$

We first consider the finite subset of the above calculus, i.e. without considering the recursion operator, and then we deal with the recursion operator.

Finite CCS

Correct and complete axiomatic presentations of several behavioural equivalences for finite CCS do exist in the literature, e.g. trace equivalence, branching bisimulation, observational congruence and testing

equivalence. All these presentations differ only for the axioms for the unobservable action. The basic set of axioms is:

$$S1. \quad E + F = F + E$$

$$S3. \quad E + \text{NIL} = E$$

$$S2. \quad E + (F + G) = (E + F) + G$$

$$S4. \quad E + E = E$$

Note that “+” is an AC operator, i.e. it is associative and commutative. If we attempt to apply the completion algorithm modulo AC [BD89], it results that some presentations admit a canonical finite term rewriting system (trace equivalence and branching bisimulation), whereas the completion of other theories (observational congruence and testing equivalence) is divergent, i.e. it results in an infinite term rewriting system [DIN90]. Therefore, it is necessary to devise an alternative way to provide a rewriting relation for the observational and testing equivalences. We consider, in particular, the observational congruence [HM85], which is characterized by the following τ -laws:

$$T1. \quad \mu.\tau.E = \mu.E$$

$$T2. \quad E + \tau.E = \tau.E$$

$$T3. \quad \mu.(E + \tau.F) + \mu.F = \mu.(E + \tau.F)$$

Our solution has been to define a rewriting strategy, *strat*, that is able to compute the normal form of a finite CCS term and verify the observational congruence of two terms without performing any completion [IN90a]. The main feature of *strat* is that it makes use of control strategies and selection criteria in order to keep some of the equations as equations, i.e. allowing expansions besides reductions, while remaining deterministic and complete.

The starting point of the strategy is the rewriting system R_{OBS} obtained by directing the axioms S3, S4, T1, T2 and T3 according to a chosen term ordering $>$.

$$R_{OBS} \quad r1. \quad E + \text{NIL} \rightarrow E$$

$$r3. \quad \mu.\tau.E \rightarrow \mu.E$$

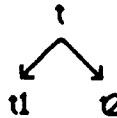
$$r2. \quad E + E \rightarrow E$$

$$r4. \quad E + \tau.E \rightarrow \tau.E$$

$$r5. \quad \mu.(E + \tau.F) + \mu.F \rightarrow \mu.(E + \tau.F)$$

R_{OBS} is correctly oriented and AC-terminating, but it is not confluent modulo AC. In fact, infinite critical pairs derive from the superposition of r2, r4 and r5 and they do not reduce to identity in R_{OBS} . Note that these rules, and those derived from critical pairs, rewrite terms by deleting one of the summands of their left-hand side.

In order to define a rewriting strategy which is complete with respect to the axiomatic presentation, we have to cope with all *peak* situations, i.e. when a term can be rewritten by means of two (or more) rules.



In the peak above, let us suppose that t can be rewritten into $t1$ and $t2$ by applying the rules r and r' , and $t1 > t2$. This means that $t1$ can be rewritten into $t2$ by applying the rule derived from the critical pair associated to the superposition of r with r' .

The definition of *strat* is based on the idea that all peaks have to be recognized and the application of the associated critical pairs has to be simulated. The strategy can be seen as composed of two phases. The first phase, *R_{OBS} -Normalization*, normalizes the input term with respect to R_{OBS} . The second phase, *Absorption*, works on the resulting term by looking for peak situations and summands to be deleted according to observational congruence. This is done by rewriting the term with T2-T3 as expansion rules (*unfolding* process which, roughly speaking, corresponds to moving up along the peak, on the left) and, as soon as possible, by deleting the redundant summands by means of R_{OBS} (*reduction* process which, roughly speaking, corresponds to deleting the top-level summand which would be deleted by applying the

rule derived from the associated critical pair). When applying such reductions, a specific redex selection criterion is used that prevents those reductions which are exactly opposite to the previous expansions by T2-T3. Another criterium is then needed to stop the unfolding and reduction steps, which are applied until there exist summands to be deleted. Finally, to obtain the normal form with respect to observational congruence, the current term is rewritten by applying the reductions opposite to the previous expansions (*folding* process which, roughly speaking, corresponds to moving down along the peak) by using a redex selection criterion that selects the smallest redexes with respect to the fixed term ordering.

This strategy can be defined as the following regular expression (r^* means repetition of the rule r until its applicability conditions are satisfied, while “;” means sequencing of rules):

strat =_{def} ROBS-Normalization; Absorption
where Absorption =_{def} (Unfolding; Reduction)*; Folding*

CCS with recursion

As soon as the recursion operator is considered, it is necessary to cope with the problem of dealing with non-terminating relations. The axiomatic presentation for observational congruence for finite state processes includes two further axioms for the recursive operator:

- R1. $\text{recX.E} = E\{\text{recX.E}/X\}$
- R2. If $F = E\{F/X\}$ then $F = \text{recX.E}$ provided X is guarded in E

Actually these are meta-axioms since they are defined in terms of a generic schema for the involved expressions. Our approach [IN90b, IN91] has been to study for an equivalent rewriting strategy in the framework defined in [DK89, DKP89, DKP90] where some conditions on infinite relations, notably *left-linearity*, and on infinite derivations, notably *fairness*, are required in order to compute the infinite normal form of a term as the limit of an infinite derivation.

Our relation is not left-linear, but we are still able to obtain normal forms as limits of fair derivations by replacing the left-linearity requirement with a *retraction* property of the supporting term algebra that allows the definition of a rewriting relation modulo an equivalence relation induced on the terms by the non-terminating rules. The retraction property means that, if a class of finite terms exists whose elements are rewritten via the non-terminating rules into the same infinite term t , it is possible to select a unique finite canonical representative of the class, $\text{CT}(t)$. Thus, an equivalence relation, denoted with $=_{\text{CT}}$, can be determined such that for any two terms t_1, t_2 , the equivalence $t_1 =_{\text{CT}} t_2$ holds if and only if $\text{CT}(t_1) = \text{CT}(t_2)$. Under this assumption we can still restrict to consider only a subset of infinite derivations, i.e. fair derivations.

Then, we go further on by focussing on those rewriting systems which admit a peculiar kind of fair derivations, i.e. *uniform* systems and *structured* fair derivations. A derivation is structured if it is possible to single out an index N such that each t_n ($n \geq N$) can be rewritten only by means of the non-terminating rules, while uniformity means that any infinite derivation with limit admits a structured derivation with the same limit. Uniformity allows the proof of the ω -confluence of an infinite relation to be split in two steps: i) to prove the confluence of the relation restricted to the finite part of fair derivations, thus retrieving all the results valid for finitely terminating rewriting relations, e.g. local confluence; ii) to prove ω -confluence only for the non-terminating rules.

These steps can be proved by properly constraining the possible interaction between the non-terminating rules and the remaining rules. In our particular case, we note that there exist recursive terms such that rewriting with the non-terminating rule \rightarrow_{R1} produces infinitely many redexes for the rule $\mu.\tau.E$

$\rightarrow \mu.E$. However, new rewriting rules can be added to prevent this situation and guarantee uniformity [IN90b]. In our case, the following axiom, *Action Prefix*, has been introduced to cope with this situation:

Ap. $\text{rec}X.\tau.E = \tau.\text{rec}X.E\{\tau.X/X\}$

As regards the very general meta-axiom R2, we replace it with a more convenient rule by specializing its application patterns. This axiom can be applied only on expressions denoting infinite trees. In fact, an expression F has to be observational congruent to an expression E containing F itself as subexpression and, obviously, this cannot be the case for finite trees. We replace R2 with the following rule, the *Collapsing Rule*, which restricts the range of application of R2 and removes bisimilar nodes other than those obtained by unfolding.

CR. Given $E \equiv \text{rec}X.E'$, if $F \equiv \text{rec}Y.F'$ is a subexpression of E' such that $E\{X/F\} = F\{Y/X\}$ and $\text{FreeVar}(E) = \text{FreeVar}(F\{Y/X\})$, then $E = E\{X/F\}$.

Now, a rewriting relation \rightarrow_{r_obs} for observational congruence of recursive (finite-state) CCS can be defined in terms of the relation \rightarrow_{strat} for finite CCS and of the rewriting rules \rightarrow_{CR} , \rightarrow_{Ap} and \rightarrow_{R1} as follows:

$$\rightarrow_{r_obs} = \rightarrow_{strat} \cup \rightarrow_{CR} \cup \rightarrow_{Ap} \cup \rightarrow_{R1}$$

where \rightarrow_{r_obs} is modulo the AC axioms and \rightarrow_{strat} is also modulo CT. This relation has been proved ω -canonical and complete with respect to the axiomatization above [IN90b].

Summing up, when dealing with the axiomatic presentations of behavioural equivalences for process algebras in a rewriting framework, requirements on the associated infinite rewriting relations such as retraction, structured derivation and uniformity, appear to be much more *natural* in our theories than the left-linearity requirement. On the other hand, the general framework defined in [DK89, DKP89, DKP90] is very suitable to study for a notion of normal forms of recursive process algebras terms.

References

- [BD89] Bachmair, L., Dershowitz, N. Completion for Rewriting modulo a Congruence, *Theoretical Computer Science* 67, (1989), 173-201.
- [CIN91] Camilleri, A., Inverardi, P., Nesi, M. Combining Interaction and Automation in Process Algebras Verification, to appear in *Proceedings TAPSOFT '91*.
- [DIN90] De Nicola, R., Inverardi, P., Nesi, M. Using the Axiomatic Presentation of Behavioural Equivalences for Manipulating CCS Specifications, in *Proc. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, (1990), 54-67.
- [DK89] Dershowitz, N., Kaplan, S. Rewrite, Rewrite, Rewrite, Rewrite, Rewrite..., *Proc. 16th Annual ACM Symposium on POPL, Austin, Texas*, (1989), 250-259.
- [DKP89] Dershowitz, N., Kaplan, S., Plaisted, D.A. Infinite Normal Forms, *Proc. 16th ICALP, Stresa, Italy, LNCS 372*, (1989), 249-262.
- [DKP90] Dershowitz, N., Kaplan, S., Plaisted, D.A. Rewrite, Rewrite, Rewrite, Rewrite, ..., draft (1990).
- [HM85] Hennessy, M., and Milner, R. Algebraic Laws for Nondeterminism and Concurrency, *Journal of ACM*, Vol.32, No.1, (1985), 137-161.
- [IN90a] Inverardi, P., Nesi, M. A Rewriting Strategy to Verify Observational Congruence, *Information Processing Letters* Vol. 35, No. 4, (August 1990), 191-199.
- [IN90b] Inverardi, P., Nesi, M. Deciding Observational Congruence of Finite-State CCS Expressions by Rewriting, I.E.I. Internal Report n° B4-10, (March 1990).
- [IN91] Inverardi, P., Nesi, M. Infinite Normal Forms for Non-linear Term Rewriting Systems, to appear in *Proceedings MFCS '91*.
- [Mil89] Milner, R. A Complete Axiomatization for Observational Congruence of Finite-State Behaviours, *Information and Computation* 81, (1989), 227-247.
- [Pr89] *Proc. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, (1990).
- [Pr90] *Proc. Workshop on Computer-Aided Verification, DIMACS Technical Report 90-31*, (June 1990).

Modeling Concurrency with AND/OR Algebraic Theories

Maria Zamfir Bleyberg
Computing and Information Sciences Department
Kansas State University, Manhattan, KS 66506
e-mail: maria@cis.ksu.edu

1 Introduction

1.1 Motivation

In the present work we treat the theory of concurrent communicating processes in an axiomatic way, following the style of J.A. Bergstra, J.W.Klop, J.C.M. Baeten, and W.P. Weijland on *Process Algebra* [1]. The basic operators of Process Algebra are \circ , $+$, and \parallel , representing the sequential composition, choice, and parallel composition, respectively. Relevant to Process Algebra is the *arbitrary interleaving* approach to the parallel operator \parallel , i.e., given atomic actions a and b , $a\parallel b = aob+boa$.

Our work is motivated by the need of an axiomatic method that covers *true concurrency*, i.e., based on *non-interleaving semantics*. We define an equational specification $T = (\Sigma, E)$, which we call an AND/OR specification, whose set E of equational axioms does not contain the interleaving axiom for \parallel , and we introduce AND/OR algebraic theories as models of T . An AND/OR algebraic theory is a particular (free) continuous many-sorted algebraic theory [8, 3], in which products and coproducts are used to represent parallelism and choice, respectively. This construction uses a base category that is a quotient of a category of sets of strings with respect to a congruence. We also introduce AND/OR nets as convenient visualization models for T . The construction of AND/OR nets has been directly influenced by CCS [6]. We show that Petri nets can also be formulated as models of T .

We also show that an AND/OR algebraic theory is a strict symmetric monoidal category [4]; a similar result was obtained by Meseguer for Petri nets [5].

1.2 Values and Continuations

Filinski, in [2], gives an elegant description of the *dual* relationship between *values* and *continuations* in a continuation-based denotational semantics. Continuations are viewed as *requests*; they are suitable for describing control structures and enable one to reason formally about termination properties of programs. Briefly, a function specified as a value abstraction $x \Rightarrow E$ describes how to transform an input value x into a result value denoted by E ; a function specified as a continuation abstraction $y \Leftarrow C$ describes how a request denoted by C is transformed into a request for the input y . Since continuations are not treated as functions, we can never explicitly apply a continuation to a value, but only substitute the current continuation with another. Nor can a function by itself be used as a continuation, because we cannot specify a destination for the result, after it has been processed by the function.

We use this dual relationship in our model. We consider that a concurrent system is built from independent computing agents capable of performing various actions (operations). The computing agents are equipped with input and output capabilities (ports) through which they communicate. A complete specification of a concurrent computing system needs to capture both the control structure and the input value

transformations when the system's actions are treated as functions.

Let us consider a category C with two kinds of finite products, \times and \parallel , and finite coproducts, $+$ and $@$, respectively, and a final object 1 . We write \circ for categorical composition. In Appendix, we give an example of such a category. The morphisms of C , such as ξ and η , represent the *atomic actions* that the concurrent system is able to perform. The categorical coproduct $\xi + \eta$ represents the *alternative composition* of ξ and η and the product $\xi \parallel \eta$ represents the *parallel composition* of ξ and η ; the categorical composition $\xi \circ \eta$ represents the *sequential composition* of ξ and η . We can represent the control structure of a concurrent computing system as a particular morphism (action) expression using the $+$, \parallel , and \circ operations. We assume that \circ binds stronger than $+$ and \parallel .

We present the dual view of values and continuations through an example. Consider the morphism composition $A \xrightarrow{\xi} B \xrightarrow{\eta} D$ that can also be written as an expression $\eta \circ \xi$. In the concrete category Set , $\eta \circ \xi(x) = \eta(\xi(x))$ shows the value transformation. The order of writing the composition of morphisms seems to be backwards. *This situation occurs because a diagram emphasizes the control transformation while an expression emphasizes the value transformation.* In our work, we use the notation $\xi(x)$ for the use of ξ as a value transformation and $y \circ \xi$ for the use of ξ as a continuation transformation (x and y are tuples). Because of the possibility of repetition of actions, we draw a distinction between events and actions. Actions label events; therefore, an event is an instance of an occurrence of its action. Composition \circ introduces a temporal constraint on events; $\eta \circ \xi$ shows that the event labeled by action η may take place only after the event labeled by action ξ has taken place.

1.3 AND/OR Specifications

Let $\Gamma = \{\alpha, \beta, \gamma, \dots\}$ be a finite set of elements, which we call *input communication names*, and $\bar{\Gamma} = \{\bar{\alpha}, \bar{\beta}, \bar{\gamma}, \dots\}$ be a finite set of complementary names, which we call *output communication names*, bijective with Γ such that $\gamma \in \Gamma$ implies $\bar{\gamma} \in \bar{\Gamma}$ and $\bar{\bar{\gamma}} = \gamma$. The elements of Γ and $\bar{\Gamma}$ represent elementary input and output capabilities, respectively, that a computing agent uses to communicate with its environment. Communication channels are established among communicating computing agents by connecting output communication names with corresponding complimentary input communication names. A communication action along a channel is regarded as an indivisible sequence *inputoutput*, where *input* and *output* are actions of the composing agents.

Let Σ be an operator domain (*signature*). Our view of a concurrent system requires the partition of the operators of Σ into three main groups: connectors (Σ_Γ), constructors (Σ_C), and actions (Σ_A).

The *action* operators of Σ_A , denoted by ξ, η, ζ, \dots , represent the indivisible actions that a communicating computing system is able to perform. We distinguish among the input, the output, and the value transformation operations. Operator 1 represents a successfully terminating action. *Receive* _{α} (d) and *Send* _{β} (d) are examples of input and output operations, respectively; *square*(2) is an example of a value transformation operation.

The *constructor* operators of Σ_C are used to construct processes out of primitive actions based on *continuations*. Σ_C contains binary operators, such as $+$, \parallel , and \circ , and unary operators, such as τ . The intuitive meaning of \circ is *sequential composition*, $+$ is *alternative composition*, and \parallel is *parallel composition*. This interpretation explains the axioms $E_1 - E_9$ in Fig.1, which are associated with E_C . Each of these equations says that the behavior of the process on its left side is the same as the behavior of the process on its right side. The τ operation has the same meaning as the *augment closure* operation of Pratt's model [7]; it weakens the notion of concurrency of two events in the absence of temporal constraints. The existing temporal constraints are preserved.

For example, $\tau(\zeta \parallel \eta \circ \xi) = \zeta \parallel \eta \circ \xi + \zeta \circ \eta \circ \xi + \eta \circ \zeta \circ \xi + \eta \circ \xi \circ \zeta + \eta \circ (\xi \parallel \zeta) + (\zeta \parallel \eta) \circ \xi$

The *connector* operators of Σ_Γ encapsulate all possible interactions among communicating computing agents. Σ_Γ includes the *sequential*, *choice*, *parallel*, and *iteration* connection operators. The required size of

this abstract makes it impossible to cover here the *connector* operations and their axioms.

The AND/OR specification $T = (\Sigma, E)$ consists of Σ as defined above and a set of axioms E for each operator group in Σ . An axiom is an equation of the form $t_1 = t_2$, where t_1 and t_2 are terms over Σ , which we call AND/OR *terms*. Among the equations of E_C we consider:

$$\begin{array}{ll}
 E_1: \xi + \eta = \eta + \xi & E_2: (\xi + \eta) + \zeta = \xi + (\eta + \zeta) \\
 E_3: \xi + \xi = \xi & E_4: \xi || \eta = \eta || \xi \\
 E_5: (\xi || \eta) || \zeta = \xi || (\eta || \zeta) & E_6: \xi \circ 1 = 1 \\
 E_7: (\xi \circ \eta) \circ \zeta = \xi \circ (\eta \circ \zeta) & E_8: (\xi + \eta) \circ \zeta = (\xi \circ \zeta) + (\eta \circ \zeta) \\
 E_9: (\xi + \eta) || \zeta = (\xi || \zeta) + (\eta || \zeta) & \text{but not } E_{10}: \xi + (\eta || \zeta) = (\xi + \eta) || (\xi + \zeta)
 \end{array}$$

Figure 1

1.4 The AND/OR Net Model

An AND/OR net is an AND/OR directed graph in which every node has associated a pair consisting of a communication name from the set of communication capabilities $\Gamma \cup \bar{\Gamma} \cup \{\lambda\}$ and an operator symbol from $\Sigma_A \cup \Sigma_\Gamma$. The communication name represents the communication capability of the AND/OR net at that node, and the operator symbol represents the action the AND/OR net is able to perform at that node. The nodes labeled by names from Γ are *input nodes*, the nodes labeled by names from $\bar{\Gamma}$ are *output nodes*, and the nodes labeled by λ are *internal nodes*. AND/OR directed acyclic graphs are of special interest; their nodes represent events and their arcs give the control structure among events. An example of an AND/OR acyclic net is given in Fig.2.

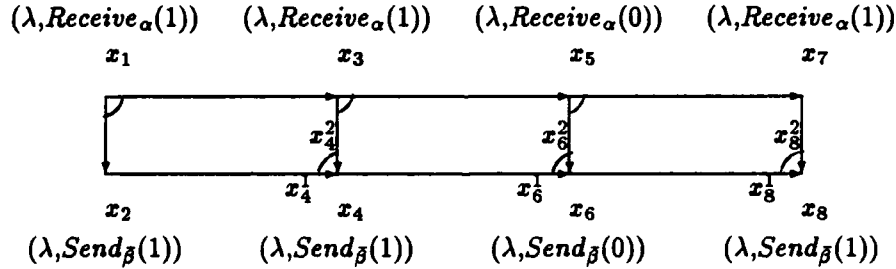


Figure 2

The nodes of an AND/OR net are of the following types:



Intuitively, a node of Out-type AND and its descendants represent the splitting of a computation into several parallel computations (a *fork* situation); a node of Out-type OR and its descendants represent the splitting of a computation into several computations, such that a single computation, out of many possible ones, is chosen nondeterministically to be performed; a node of In-type AND and its predecessors represent parallel computations that join together again on a single task (a *join* situation); a node of In-type OR and its predecessors represent several computations competing for exclusive access to a task (mutual exclusion). We interpret as continuations the variables x_1, \dots, x_8 that we use to identify the nodes of the AND/OR net in Fig.2 and associate an equation to each node in the following way

$$\begin{aligned}
x_1 &= (x_2 || x_3) \circ \text{Receive}_\alpha(1) \\
x_2 &= x_4^1 \circ \text{Send}_\beta(1) \\
x_3 &= (x_4^2 || x_5) \circ \text{Receive}_\alpha(1) \\
x_4^1 || x_4^2 &= x_6^1 \circ \text{Send}_\beta(1)
\end{aligned}$$

$$\begin{aligned}
x_5 &= (x_6^2 || x_7) \circ \text{Receive}_\alpha(0) \\
x_6^1 || x_6^2 &= x_8^1 \circ \text{Send}_\beta(0) \\
x_7 &= x_8^2 \circ \text{Receive}_\alpha(1) \\
x_8^1 || x_8^2 &= \text{Send}_\beta(1)
\end{aligned}$$

The solution of this system of equations represents the behavior (the control structure) of the AND/OR net in Fig.2. In an ω -continuous AND/OR algebra such a system of equations has a least fixed-point solution. All this is clearer in an AND/OR algebraic theory framework in which the above system of equations is a morphism. Substitution (the categorical composition in an algebraic theory) gives an *operational semantics* for the control structure.

Once the control structure of a concurrent system is defined, it can be used as a value transformation (in category Set) that maps the inputs to the system into outputs.

2 An Example

We illustrate the power of our approach to concurrency by specifying a channel. Variants of this example have been treated in both Process Algebra [1] and Modeling Concurrency with Partial Orders [7].

A communication channel between endpoints α and β is a process in which events happen in pairs. A pair consists of a transmission and a receipt of a data element from one end to the other end of the channel. Data elements can be passed in both directions simultaneously.

Assume we have a communication channel as given in Fig.3, which has α and β as input ports and $\bar{\alpha}$ and $\bar{\beta}$ as output ports. Data elements are 0 and 1 ($d \in \{0,1\}$).

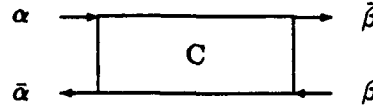


Figure 3

We consider the following atomic actions: $\text{Receive}_\alpha(d)$ for receiving d at port α
 $\text{Send}_\alpha(d)$ for sending d at port $\bar{\alpha}$
 $\text{Receive}_\beta(d)$ and $\text{Send}_\beta(d)$

A specification of the channel $C_{\alpha\bar{\beta}}$ from α to $\bar{\beta}$ is given by the following recursive equation over theory \mathcal{T} :

$$C_{\alpha\bar{\beta}}^1 = \tau(C_{\alpha\bar{\beta}}^2 || \text{Send}_\beta^1(d) \circ \text{Receive}_\alpha^1(d)) + 1$$

The role of a superscript, such as 1 and 2, is to "stamp" each iteration with the repetition number; iterations create time constraints on the events of a process that are labeled by the same action.

The behavior of $C_{\alpha\bar{\beta}}$ for the sequence 1101 is given by solving this equation, which is the AND/OR term

$$\tau(((\text{Send}_\beta^4(1) \circ \text{Receive}_\alpha^4(1) || \text{Send}_\beta^3(0) \circ \text{Receive}_\alpha^3(0) || \text{Send}_\beta^2(1) \circ \text{Receive}_\alpha^2(1) || \text{Send}_\beta^1(1) \circ \text{Receive}_\alpha^1(1))$$

Similarly, a specification of the channel process $C_{\beta\alpha}$ is given by the following equation

$$C_{\beta\alpha}^1 = \tau(C_{\beta\alpha}^2 || \text{Send}_\alpha^1(d) \circ \text{Receive}_\beta^1(d)) + 1$$

The bidirectional channel has the specification $C^1 = \tau(C_{\alpha\bar{\beta}}^1 || C_{\beta\alpha}^1)$

Fig.2 shows the representation of the the channel behavior (for the sequence 1101) as an AND/OR net.

Suppose now that we need to specify an **unreliable channel**, which may send on \perp instead of 0 or 1. Let *alter* represent an alteration action; it is $alter(0) = \perp$ and $alter(1) = \perp$ as a function. Let *pass* represent the identity action, i.e., $pass(0) = 0$ and $pass(1) = 1$ as a function. A specification of the unreliable channel $UC_{\alpha\bar{\beta}}$ from α to $\bar{\beta}$ is given by the following recursive equation over theory \mathcal{T} :

$$UC_{\alpha\bar{\beta}}^1 = \tau(UC_{\alpha\bar{\beta}}^2 \parallel Send_{\bar{\beta}}^1(d) \circ (alter^1(d) + pass^1(d)) \circ Receive_{\alpha}^1(d) + 1$$

We present a superficial comparison between our approach to concurrency and other models of concurrency by showing in Fig.4 and Fig.5 the behavior of the channel in Fig.3 using the Pratt's model [7] and the Petri net model, respectively.

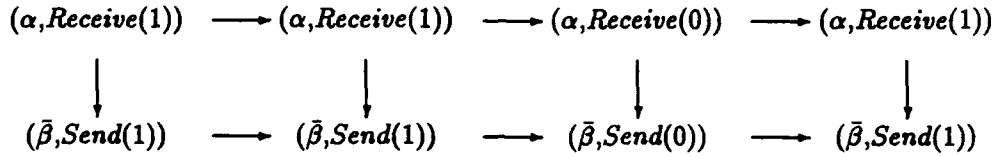


Figure 4

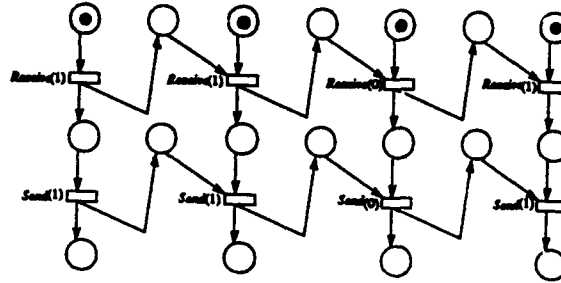


Figure 5

References

- [1] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [2] A. Filinski. Declarative continuations: An investigation of duality in programming language semantics. *Mathematical Structures in Computer Science*, 1, 1991.
- [3] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24, Jan. 1977.
- [4] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.
- [5] J. Meseguer and U. Montanari. Petri nets are monoids. In *LNCS*. Springer Verlag, 1989.
- [6] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [7] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.
- [8] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Notes on algebraic fundamentals for theoretical computer science. In *Foundations of Computer Science, Part 2: Languages, Logic, Semantics*, Amsterdam, 1979. Addison-Wesley.

A An Example Category

It is known that we can describe a function via a set. For function $f: R^m \rightarrow P^n$, the set

$$\text{graph}(f) = \{(x, f(x)) \mid x \in R^m\}$$

is the graph of f . We write here a tuple $x \in R^m$ as a string $x_1 \dots x_m$; similarly, a tuple $y \in P^n$ is a string $y_1 \dots y_n$.

Given $S = RUP$, the category Set_S consists of sets representing function graphs over S as *objects* and a pre-order relation \subseteq on these sets, which we will introduce below, as *morphisms*.

First, we introduce the "less general" pre-order \leq on strings over S :

$$x \leq y \text{ iff } y \text{ is a substring of } x.$$

For example, $00 \leq 0$ and $13 \leq 3$. Then,

$$(x, y) \leq (x', y') \text{ iff } x \leq x' \text{ and } y \leq y'.$$

By definition, $A \subseteq B$ iff

1. $A \subseteq B$ and we say that A is "less informed than" B or,
2. for every $a \in A$ there exists a $b \in B$ such that $a \leq b$.

Products and *coproducts* are defined in Set_S in the following way:

1. $A + B$ is the "least upper bound" (*lub*) of A and B
2. $A \times B$ is the "greatest lower bound" (*glb*) of A and B
1. $A @ B$ is the "disjoint union" of A and B
2. $A \parallel B$ is the "cartesian product" of A and B

Example. Let $S = \{0, 1, 2, 3, 4\}$. Consider the sets $A = \{(0, 1)\}$, $B = \{(0, 1), (1, 3)\}$, and $C = \{(2, 4)\}$. It is easy to verify that the axioms E_1 to E_9 are satisfied and E_{10} is not:

$$\begin{aligned} A \parallel C &= \{(02, 14)\}, B \parallel C = \{(02, 14), (12, 34)\}, \text{ and } A \parallel C + B \parallel C = \{(02, 14), (12, 34)\} \\ A + B &= \{(0, 1), (1, 3)\} \text{ and } (A + B) \parallel C = \{(02, 14), (12, 34)\} \end{aligned}$$

In Set_S , the final object 1 is the "most general" and "most informed" set. There is no initial object.

The category Set_S is an example of a category with two kinds of finite products, \times and \parallel , and finite coproducts, $+$ and $@$, respectively, and a final object 1 .

TYPE CONSISTENCY CHECKING FOR CONCURRENT INDEPENDENT PROCESSES (CIP)

by

Dr. Aurel Cornell
Computer Science Department,
Brigham Young University, Utah.

1 INTRODUCTION

The programming paradigm where a program is "composed of independent units" raises the question of how are the units going to communicate to each other, how is the consistency of the program being checked and guaranteed in a parallel and distributed system. One solution to this problem is to use type signatures like in Linda [GELE 85] for variables through which the modules will communicate. The way signatures are used in LINDA does not guarantee that the system will not block, and that a correct data delivery will take place, because the user is responsible to define for each variable the correct signature. A better approach to this problem is to have the compiler generate the signatures. This paper presents how the channel correct data delivery and continuous data flow has been implemented and guaranteed in the CIP compiler by using signatures that are generated by the compiler.

Communicating Independent Processes (CIP) [CORN 89] is a message passing language based on Modula-2 [WIRT 83] and C.A.R. Hoare's Communicating Sequential Processes (CSP) [HOAR 78] with its input and output functions. CIP is a concurrent programming language which consists of two main types of units: processes and channels. The process is the main computational entity; it does not share data and can only communicate with other processes through input/output (I/O) commands which either read or write data to or from a channel. A channel can be thought of as an intelligent buffer through which two or more processes communicate synchronously. Thus a process can write to a channel and continue to execute afterwards even when other processes are not ready to read that data. The complete description of the CIP language is presented in [HAT 88].

2 SIGNATURE CONSISTENCY CHECKING

In CIP channels represent the means through which independent processes communicate. A channel has to be consistent in order to avoid an incorrect data delivery or deadlocking.

The formal definition of consistency is divided into two parts: 1) *a channel must deliver the correct data for every request and 2) a channel has to guarantee a continuous data flow.* This means that for every type of data written to a channel by a process there is another process connected to that channel which can read that data, and for every read request for a specific type of data there is a process that can write that data to that channel.

CIP uses static and dynamic signatures to guarantee channel consistency. A Producer and Consumer example is used to illustrate the correct data delivery (Fig. 1, case A, and B). A signature is a unique identifier generated by the compiler for each type declared in a library. A variable defined, based on a library type, automatically receives the same signature as its type. The variables that are sent as a message from one process to another -via a channel- must all have a signature. This aspect is checked and guaranteed by the compiler. All the other variables that are defined and used only inside of a process do not need signatures, hence for these variables the compiler will not generate signatures.

A variable that is written in a channel loses its identity and it is represented by the tuple (v, s) where v is the value and s is the signature of the variable. The only way such a variable is retrieved from the channel

is through its signature: a process connected to the channel requests a variable with the signature s.

2.1 CORRECT DATA DELIVERY WITH STATIC SIGNATURES.

In Figure 1 case A the producer writes two type of data, s and t, both of type real, but with two different signatures produced by the compiler for the types Time and Space. Because the signatures are different the correct data delivery is guaranteed by the compiler. This means that no matter in what order (VAL(t), SIG(Time)) and (VAL(s), SIG(Space)) are written in the channel, the consumer will always read a time value for time and a space value for space.

In Figure 1 case B a (VAL(a), SIG(INT1)) is written in the channel and a (VAL(b), SIG(INT2)) is requested, which will produce a channel incomplete error because INT1 and INT2 do not have the same signature.

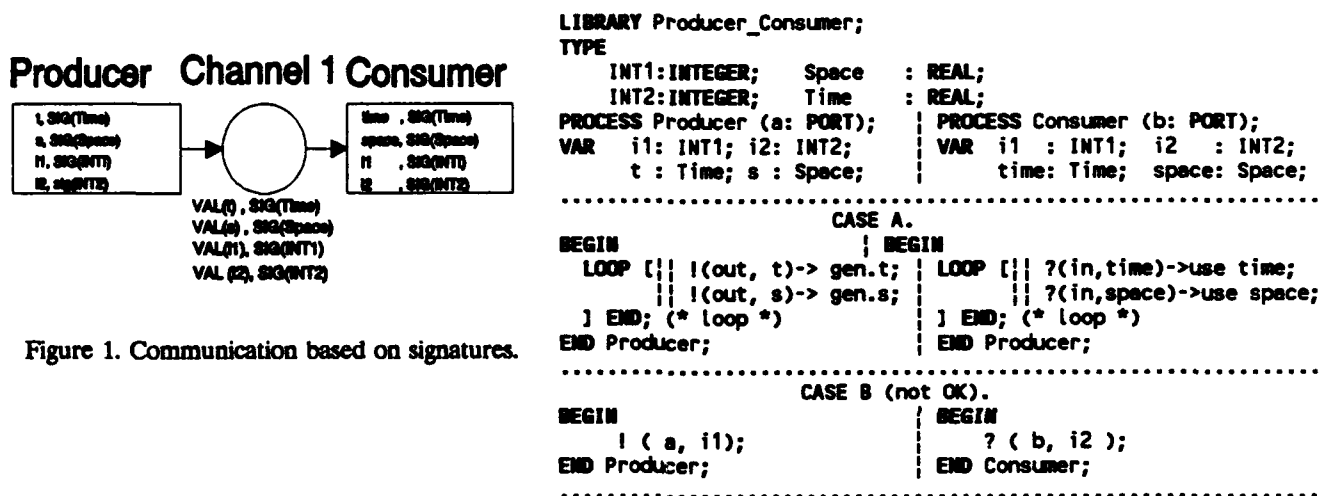


Figure 1. Communication based on signatures.

2.2 CORRECT DATA DELIVERY WITH DYNAMIC SIGNATURES.

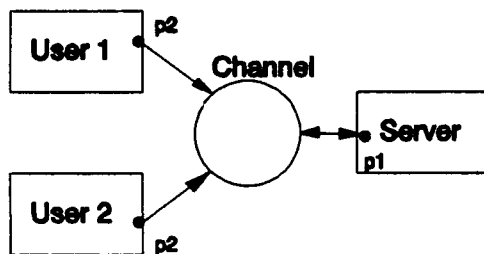
The signature as described above is insufficient when a server process can service more than one process and the processes requesting the service expect a specific response back from the server. This is similar to a function call in a sequential program.

The example below illustrates the need for a process to uniquely identify its data at run time. Figure 2 shows the configuration of the example. Processes user 1 and user 2 both will send data of type REQUEST, but since the server sees the request as identical, it cannot determine to whom to send the response. It would simply write data of type RESPONSE back into the channel. Thus user 1 could get the response message which was intended for user 2 and vice versa.

The dynamic signature has been created to allow a process to uniquely identify a specific type of data at run time. This can be sent to another process which then can identify the type of data that is to come or to be sent back. Two new intrinsic procedures, NEWSIG and ASSIGNSIG, were created to manipulate dynamic signature variables. NEWSIG will assign a unique identifier to a variable declared as a dynamic type. ASSIGNSIG will assign the dynamic signature and the data from one dynamic variable to another variable of the same exact type. If the normal assignment (:=) is used, only the data is transferred.

The use of dynamic signatures can be best described using postal terminology. When a company sends a bill, it normally encloses a self-addressed envelope because it removes the possibility of the check being sent to the wrong address. The customer places the check in the envelope and mails it back. When a user process sends a request (a bill), it encloses a dynamic variable (self-addressed envelope). The server process assigns the dynamic variable (places the check in envelope) and sends it back through the channel (mails it back).

Figure 3 illustrate how the return-addressed envelope works. The user process assigns the dynamic field in the record an unique signature using the NEWSIG procedure. The user then sends the record to the server process and within this record is the dynamic variable through which the server returns the answer.



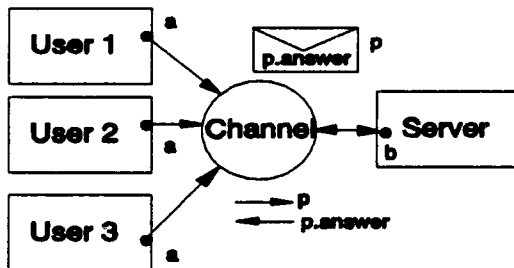
```

FROM library IMPORT REQUEST, RESPONSE;
PROCESS Server (p1: PORT);
VAR request : REQUEST;
    response: RESPONSE;
BEGIN
  LOOP
    ?(p1 ,request);...
    !(p1 ,response);...
  END;
END Server;

PROCESS User (p2: PORT);
VAR request : REQUEST;
    response: RESPONSE;
BEGIN
  LOOP
    !(p2, request);...
    ?(p2, response);...
  END;
END User;

```

Figure 2 Two users connected to one server.



```

TYPE Problem = RECORD
  datain: Data;
  answer: DYNAMIC Answer;
END;

User1, User2, and User3
PROCESS User(a: PORT);
VAR p: Problem;
BEGIN
  NEWSIG(p.answer);
  !(a, p); ...
  ?(a, p.answer)
  ...
END User;

Server
PROCESS Server(b: PORT)
VAR p: Problem;
BEGIN
  ?(b, p);
  ... calculate answer..
  !(b, p.answer);
  ...
END Server;

```

Figure 3 Communication based on dynamic signatures.

2.3 CHANNEL COMPLETENESS

Channel continuous data flow checking is done by the compiler when a CONFIGURE statement is compiled. The general syntax of a configuration is shown below.

```

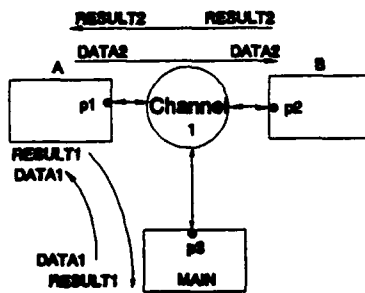
VAR config_variable : CONFIG;
config_variable := CONFIGURE (
  NEWCHANNEL (constant_expression),
  CONNECT (port_variable, channel_number) {, CONNECT (port, channel_number) }
  {,NEWPROCESS process_name | process_variable (channel_number
  {,channel_number}),}
);

```

By allowing processes and channels to be only created within a configuration, the compiler can determine if all the channels are complete. If processes and channels could be created separately and then attached together, it would be impossible to check for continuous data flow.

The configuration presented in figure 4a is created in CIP by a CONFIGURE statement like the one in Figure 4b. To guarantee continuous data flow the compiler builds a table like the one in Table 1.

For each data type (signature) that is seen by a channel there must exist at least one pair of (!, ?) commands, otherwise data flow through the channel is interrupted. The compiler flags this type of errors which otherwise would be very difficult to be detected at run time.



```
VAR c : CONFIG;
```

```
c := CONFIGURE (
  NEWPROCESS (B, 1) ,
  NEWPROCESS (A, 1),
  CONNECT (p3, 1)
);
```

Figure 4. Configuration generation.

A) The Processes and the channel of the configuration.

B) The Program that generates the configuration.

PORTS	Signature							
	DATA 1		DATA 2		RESULT 1		RESULT 2	
	In	out	In	out	In	out	In	out
A - p1	X			X		X	X	
B - p2			X					X
Main - p3		X			X			
Channel Data Continuity	OK		OK		OK		OK	

Table 1 Consistency checking table.

3 CONCLUSION

Consistency in CIP deals with two different aspects of the channel. First, the channel must deliver the correct data to the processes. The static and dynamic signatures are used to handle this problem. They allow a channel to uniquely identify the different types of data being requested. Second, the processes connected to a channel must have matching input/output commands for each signature used by the channel. This is enforced by only allowing processes and channels to only be created in configurations. This permits the compiler to know when to check the completeness of each configuration.

4 REFERENCES

- CORN 89 Cornell A., "Parallel Programming in CIP", "Proceedings of the 36 ISMM Conf. on Mini and Microcomputers and their Applications", pp 542-548, Barcelona, Spain, 1989.
- GELE 85 Gelernter, D., "Generative Communication in Linda", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985.
- HOAR 78 Hoare, C.A.R., "Communicating Sequential Processes". Communications of the ACM, Vol. 21, No. 8 (August 1978).
- HATC 88 Hatch, Rex, "The CIP Language", Report, Computer Science Department, BYU, 1988.
- WIRT 83 Wirth, N., Programming in MODULA-2, New York, Springer-Verlag 2nd. Edition, 1983.

Theory of Algebraic Module Specification including Behavioural Semantics, Constraints and Aspects of Generalized Morphisms

Hartmut Ehrig, Michael Baldamus, Felix Cornelius

Technical University Berlin

(Germany)

Fernando Orejas

Catalan University of Technology

(Spain)

February 1991

CONTENTS

1. Introduction
 2. Specification Logics
 3. Specification Logics with Constraints
 4. Abstract Module Specifications
 5. Behavioural Module Specifications
 6. Behavioural Module Specifications with Constraints
 7. Example (Behavioral Case of Airport Schedule Modules)
 8. References
- Appendix: Generalized Morphisms

ABSTRACT

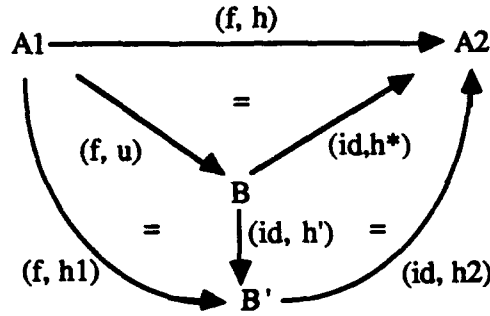
The theory of algebraic module specifications and modular systems developed mainly in the framework of equational algebraic specifications is shown to be almost independent of the underlying kind of signatures and axioms. In fact the theory can be formulated entirely categorically in the abstract framework provided by the notion of specification logics. To obtain the desired results pushouts on the syntactic side and amalgamations or extensions as well as the existence of some free constructions on the semantic side rather have to be assumed than are shown for a concrete specification formalism. The general framework is applied to module specifications with behavioural semantics and constraints. In the appendix we study generalized morphisms and their relationship to amalgamation.

1. INTRODUCTION

The importance of decomposing large software systems into smaller units, called modules, to improve their clarity, facilitate proofs of correctness, and support reusability has been widely recognized within the programming and software engineering community. For all stages within the software development process modules resp. module specifications are seen as completely self-contained units which can be developed independently and interconnected with each other.

An algebraic module specification MOD as developed in [EW 85], [BEP 87] and [EM 90] consists of four components

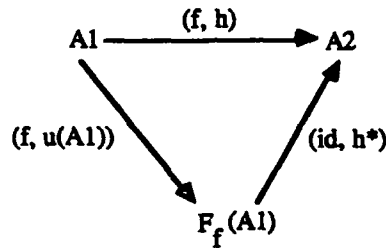
• (f, h_1) into a generalized and a standard morphism there is a unique standard morphism (id, h') such that the following diagram commutes



Remark: Canonical factorizations are unique up to isomorphism.

A.4 LEMMA (Canonical Factorization)

Given a specification logic SL with free constructions then each generalized morphism $(f, h): A_1 \rightarrow A_2$ in GMSL has a canonical factorization



where $u(A_1): A_1 \rightarrow V_f F_f(A_1)$ is the universal morphism for A_1 and the free functor F_f and $h^*: F_f(A_1) \rightarrow A_2$ the induced morphism by h .

Remark: This canonical factorization is a generalization of Higgins construction in [Hig 64] resp. [Ru 90].

Proof

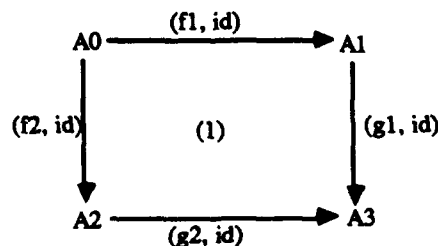
Follows immediately from the universal properties of the universal morphism $u(A_1)$ of the free functor F_f .

A.5 FACT (Pushout Properties of Amalgamation)

An amalgamation $A_1 +_{A_0} A_2$ in a specification logic SL (see 2.5) is a pushout object (of model-identical generalized morphisms) in the category GMSL.

Proofidea

Given an amalgamation $A_3 = A_1 +_{A_0} A_2$ based on a pushout diagram in ASPEC as shown in 2.5 we obtain the following diagram (1) of model-identical generalized morphisms in GMSL. The pushout#



properties of (1) in GMSL can be shown using the pushout properties of the corresponding ASPEC-diagram

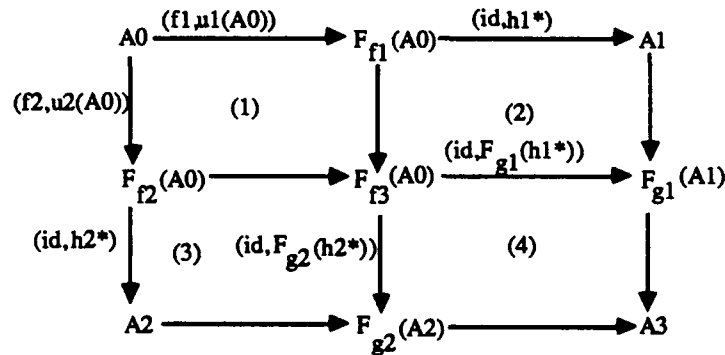
and the amalgamation properties of A_3 w.r.t. morphisms (see 2.5.3) and the Amalgamation Lemma in [EM 85]).

A.6 FACT (Pushouts in GMSL and Generalized Amalgamation)

Given a specification logic SL with free constructions, pushouts in $ASPEC$ and pushouts in all model categories $Catmod(ASPEC)$ for all abstract specifications $ASPEC$ in $ASPEC$. Then the category $GMSL$ has pushouts. The pushout object $A_3 = A_1 +_{A_0} A_2$ can be interpreted as a generalized amalgamation where the model-identities $V_{f_1}(A_1) = A_0 = V_{f_2}(A_2)$ are replaced by generalized morphisms from A_0 to A_1 and A_2 .

Proofidea

Given generalized morphisms $(f_1, h_1): A_0 \rightarrow A_1$ and $(f_2, h_2): A_0 \rightarrow A_2$ we construct the canonical factorization (see A.4) of these morphisms leading to the following pushout diagrams (1) - (3) in $GMSL$ and (4) in $Catmod(ASPEC3)$ which is also a pushout in $GMSL$. The morphisms g_1 and g_2 are given by the pushout of f_1 and f_2 in $ASPEC$ (see 2.5) and $f_3 = g_1 \circ f_1 = g_2 \circ f_2$



□

MOD:

PAR	EXP
IMP	BOD

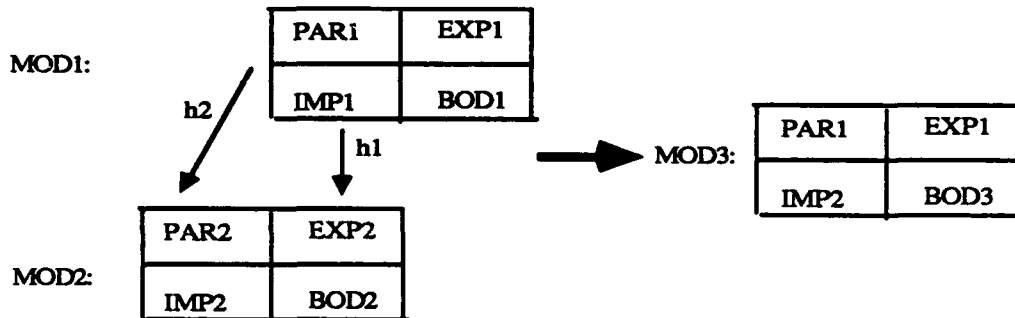
which are given by four algebraic specifications in the sense of [Zi 74], [TWW 78/82] and [EM 85]. The export EXP and the import IMP represent the interfaces of a module while the parameter PAR is a part common to both import and export and represents a part of the parameter of the whole system. These interface specifications PAR, EXP, and IMP are allowed to be algebraic specifications with constraints in order to be able to express requirements for operations and domains in the interfaces by suitable logical formalisms. The body BOD, which makes use of the resources provided by the import and offers the resources provided by the export, represents the constructive part of a module.

The semantics of a module specification MOD as above is given by the loose semantics of the interface specifications PAR, EXP, and IMP, a "free construction" from import to body algebras, and a "behavior construction" from import to export algebras given by restriction of the free construction to the export part.

A module specification is called (internally) correct if the free construction "protects" import algebras and the behavior construction transforms import algebras satisfying the import constraints into export algebras satisfying the export constraints.

Basic interconnection mechanisms applied to module specifications are composition, union, and actualization. In this paper we only consider

Composition



where h1 and h2 are passing morphisms and $BOD3 = BOD1 +_{IMP1} BOD2$ is a pushout construction.

As main results for module specifications we have shown in [EM 90] that the basic interconnection mechanisms are operations on module specifications which are preserving correctness and which are compositional w.r.t. the semantics. This means that correctness of modular system specification can be deduced from correctness of its parts and its semantics can be composed from that of its components. Moreover, there are nice compatibility results between these operations which can be expressed by associativity, commutativity and distributivity results (see [EM 90]). The relationship between algebraic module specifications and program modules is discussed in [LEFJ 91].

In this paper we show how the basic theory of algebraic module specifications can be extended from equational algebraic specifications to several other kinds of specifications including conditional algebraic specifications, different kinds of behavioral specifications [NO 88] and projection specifications for combined data type and process specifications [GR 89]. For this purpose we review in Section 2 the notion of a specification logic which is also used in [Ehr 89b], [EPO 89] and [Ma 89] and is closely related to institutions in the sense of [GB 84]. In section 3 we extend a specification logic by constraints in the sense of [Ehr 89a] and [EM 90]. We show how results concerning pushouts, amalgamation and extension can be extended from the case without to the case with constraints [Bal 90]. Assuming to have pushouts, free constructions, amalgamation and extension we are

able to extend some basic constructions and results of algebraic module specifications to arbitrary specification logics in section 4. In section 5 and 6 the theory is applied to module specifications with behavioral semantics in the sense of [NO 88] and behavioral semantics with constraints in the sense of [Ehr 89a] and [Cor 90]. In the appendix we introduce generalized morphisms in a specification logic which include general morphisms in the sense of Higgins [Hig 64] and Rus [Ru 90] as special cases and study the relationship between amalgamation and pushouts of generalized morphisms.

2. SPECIFICATION LOGICS

In this section we introduce the notion of a specification logic - also used in [Ma 89] and [EPO 89] - and formulate some basic properties like pushouts, free constructions, amalgamation and extension in this framework.

2.1 DEFINITION (Specification Logic)

A specification logic SL is a pair $(ASPEC, Catmod)$ where $ASPEC$ is a category of abstract specifications and $Catmod: ASPEC^{op} \rightarrow CATCAT$ is a functor, that associates to every specification $ASPEC$ in $ASPEC$ its category of models $Catmod(ASPEC)$ which is an object in the "category" $CATCAT$ of all categories.

Remarks

1. The functor $Catmod$ assigns to each abstract specification $ASPEC$ a category $Catmod(ASPEC)$ which usually consist of all models satisfying $ASPEC$, together with their associated morphisms. Actually there is a close relationship to institutions in the sense of [GB 84] and [ST 84]. This is discussed in more detail in the downward remark 2.2.3. Note that there is no explicit satisfaction relation contained in the notion of a specification logic and thus certain semantical and semantical properties have to be required. These are defined in 2.3 - 2.6 below.
2. If $f: ASPEC1 \rightarrow ASPEC2$ is a morphism in $ASPEC$ then $Catmod(f): Catmod(ASPEC2) \rightarrow Catmod(ASPEC1)$ is usually called the forgetful functor associated to f and it is usually denoted by V_f . Note that the functor $Catmod$ is contravariant in $ASPEC$ (denoted by $ASPEC^{op}$) such that all arrows are reversed.

2.2 EXAMPLES (Specification Logics)

1. The equational specification logic $EQSL = (SPEC, Catmod)$ consists of the category $SPEC$ of equational algebraic specifications and the functor $Catmod$ assigns to each specification $SPEC$ in $SPEC$ the category $Alg(SPEC)$, i.e. $Catmod(SPEC) = Alg(SPEC)$. Replacing equations by conditional equations we obtain the conditional equational specification logic $CEQSL$.
2. For each institution $INST = (SIGN, Sen, Mod, \models)$ the corresponding specification logic $SL(INST) = (ASPEC, Catmod)$ is defined in the following way: We take $ASPEC$ to be the category of all presentations (SIG, E) , where SIG is an object in $SIGN$ and E a subset of all sentences $Sen(SIG)$, and all presentation morphisms and $Catmod(SIG, E)$ to be the full subcategory of $Mod(SIG)$ satisfying all sentences E . Using closed sets E of sentences we can replace presentations by theories. Vice versa, each specification logic $SL = (ASPEC, Catmod)$ defines a trivial institution $INST0 = (ASPEC, \emptyset, Catmod, \emptyset)$ with empty sentence functor and empty satisfaction relation.
3. The behavioral equational specification logic $BEQSL = (BSPEC, BCatmod)$ consists of the category $BSPEC$ of behavioral specifications and the functor $BCatmod$ assigning to each behavioral specification $BSPEC$ the category $Beh(BSPEC)$ (see [NO 88], [ONE 89]). Behavioural specification cannot be seen as an

institution in the obvious way because the behavioural satisfaction of equations violates the satisfaction condition (see fact 5.2.2). Hence behavioural specification is an important example of a specification formalism which can be studied in the framework of specification logics but not in the framework of institutions. More details are given in section 5.

4. A related example of a specification formalism which cannot be seen as an institution in the obvious way is the specification logic VIEWSL of behaviour specifications and view morphisms (see fact 5.2.2 as well). These are slightly more general than behaviour morphisms. VIEWSL is also going to be discussed in section 5.

5. In section 3 we will show how to enrich an arbitrary specification logic SL by constraints leading to a new specification logic SLC which inherits some of the properties of SL. In this way we obtain from examples 1-3 the corresponding specification EQSLC, CEQSLC, BEQSLC, and VIEWSLC with constraints. The last two of them will be studied and applied in section 6.

6. The projection specification logic PROSL = (PROSPEC, PCatmod) consists of the category PROSPEC of projection specifications and the functor PCatmod assigns to each projection specification PROSPEC the category $\text{Cat}_{\text{compl,sep}}(\text{PROSPEC})$ of all complete and separated projection algebras satisfying PROSPEC (see [EPBRDG 87] and [GR 89]).

7. In addition to the specification logics mentioned above, there are several other examples including different kinds of axioms, like universal Horn or full first order axioms, order-sorted signatures and constraints on the syntactical level, and on the semantical level different kinds of algebras and structures, like partial or continuous algebras or models of first-order logic. The first-order specification logic FOSL, for example, has first order signatures and axioms on the syntactical and corresponding models on the semantical level.

2.3 DEFINITION (Free Constructions and Persistency)

1. A specification logic SL has free constructions if for every specification morphism $f: \text{ASPEC1} \rightarrow \text{ASPEC2}$ in ASPEC there is a functor $F_f: \text{Catmod}(\text{ASPEC1}) \rightarrow \text{Catmod}(\text{ASPEC2})$ which is left adjoint to V_f .
2. F_f (and, in general, any functor $F: \text{Catmod}(\text{ASPEC1}) \rightarrow \text{Catmod}(\text{ASPEC2})$) is said to be strongly persistent if $V_f \circ F_f = \text{ID}$. Given a strongly persistent free functor F_f it can be shown that each identity $\text{id}_A: A \rightarrow A = V_f(F_f(A))$ has the universal property (see [Bal 90]). We therefore assume $u(A) = \text{id}_A$ for all A whenever F_f is strongly persistent.
3. A specification morphism f is called strongly liberal if there is a strongly persistent free functor F_f left adjoint to V_f .

Remark

Free constructions have been used at the model level to give semantics to parameterized specifications. Equational specification logic EQSL and CEQSL behavioral equation specification logic BEQSL and projection specification logic PROSL have free constructions (see [EM 85], [NO 88] and [GR 89]). In contrast the specification logics EQSLC, CEQSLC and BEQSLC to be introduced below do not have free constructions in general, although specific specification morphisms are strongly liberal.

2.4 DEFINITION (Pushouts)

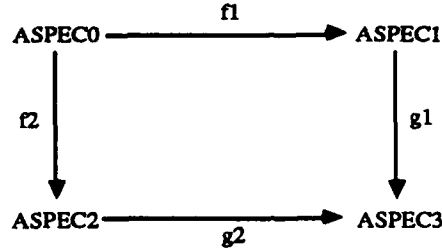
A specification logic $\text{SL} = (\text{ASPEC}, \text{Catmod})$ has pushouts if the category ASPEC has pushouts.

Remark

Pushouts are the operations, at the specification level, used to combine specifications. Essentially, if we want to put together two specifications ASPEC1 and ASPEC2 having a common subspecification ASPEC0, the pushout, ASPEC3, of ASPEC1 and ASPEC2 with respect to ASPEC0 would provide the right combination. Almost all specification logics of practical interest have pushouts (see [EM 85] for more detail).

2.5 DEFINITION (Amalgamations)

A specification logic SL has amalgamations, if for every pushout diagram in ASPEC



we have:

1. For every $A_i \in \text{Catmod}(\text{ASPEC}_i)$ ($i = 0, 1, 2$) such that

$$V_{f1}(A1) = A0 = V_{f2}(A2)$$

there is a unique $A3 \in \text{Catmod}(\text{ASPEC}_3)$, called amalgamation of $A1$ and $A2$ via $A0$, written

$$A3 = A1 +_{A0} A2,$$

such that we have

$$V_{g1}(A3) = A1 \text{ and } V_{g2}(A3) = A2.$$

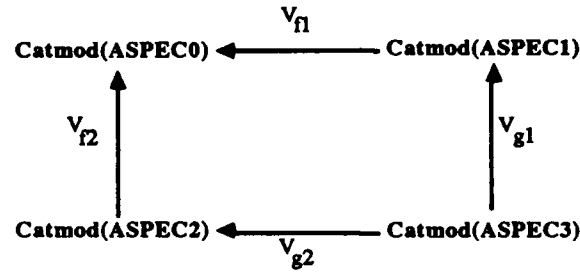
2. Conversely, every $A3 \in \text{Catmod}(\text{ASPEC}_3)$ has a unique decomposition

$$A3 = V_{g1}(A3) +_{V_{g1} \circ f1(A3)} V_{g2}(A3).$$

3. Similar properties to 1 and 2 above hold if we replace objects A_i by morphisms h_i in $\text{Catmod}(\text{ASPEC}_i)$ for $i = 0, 1, 2, 3$ leading to a unique amalgamated sum of morphisms $h3 = h1 +_{h0} h2$ with $V_{g1}(h3) = h1$ and $V_{g2}(h3) = h2$.

Remarks

1. In fact, properties 1 to 3 above are equivalent to the fact that the following diagram is a pullback in the category CATCAT of categories



2. Amalgamation allows to define the semantics of a combined specification purely on the semantic level as the class of all possible amalgamations of the specifications which are combined. If amalgamations do exist, however, this class is the same as $\text{Catmod}(\text{ASPEC3})$. The specification logics EQSL, CEQSL and PROSL have amalgamation but behavioral equational specification logic, for example, has not. In particular, in the latter case, not all but at least pushouts satisfying the "observation preserving property" [ONE 88] have associated amalgamation. See section 5 for more details.

2.6 DEFINITION (Extensions)

A specification logic SL has extensions if for every pushout diagram in ASPEC as given above (see 2.5), if

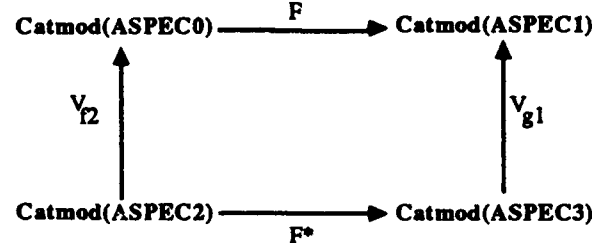
$$F: \text{Catmod}(\text{ASPEC0}) \rightarrow \text{Catmod}(\text{ASPEC1})$$

is a strongly persistent functor then there is a strongly persistent functor

$$F^*: \text{Catmod}(\text{ASPEC2}) \rightarrow \text{Catmod}(\text{ASPEC3}),$$

called extension of F via f_2 , such that:

1. the following diagram commutes



2. If F is a strongly persistent free functor with respect to f_1 then F^* is a strongly persistent free functor with respect to g_2 .

Remarks

Extension is often needed if we want to have compatibility between the semantics of certain specification building operations defined at the specification and at the model levels. The Equational and the conditional specification logic do have extensions (see [EM 85]) and so has behavioral equational specification logic under the same conditions under which it has amalgamations. In the latter case, if we consider view specification morphisms instead of behavioral specification morphisms, then behavioral equational specification logic does not have amalgamations, but a restricted form of extensions (see section 5). However, in most examples it is easier to verify amalgamations than extensions and the following results show that amalgamations are sufficient to have extensions.

2.7 THEOREM (Extension by Amalgamation)

If a specification logic $SL = (\text{ASPEC}, \text{Catmod})$ has amalgamations then SL has extensions.

Proofidea

Given a pushout as in 2.5 and assume that $F: \text{Catmod}(\text{ASPEC0}) \rightarrow \text{Catmod}(\text{ASPEC1})$ is a strongly persistent functor. We define $F^*: \text{Catmod}(\text{ASPEC2}) \rightarrow \text{Catmod}(\text{ASPEC3})$ for all objects A_2 in $\text{Catmod}(\text{ASPEC2})$ by the following amalgamation (see 2.5):

$$F^*(A_2) = A_2 +_{A_0} F(A_0)$$

with $A_0 = V_{f_2}(A_2)$ and similar for morphisms. This amalgamation is well-defined because we have $V_{f_1} \circ F(A_0) = A_0$ by strongly persistency of F .

The remaining part of the proof is given in [EM 85] already almost entirely categorically for the specification logic EQSL. The only difference in the abstract case is that our convention $u(A) = \text{id}_A$ for a strongly persistent free functor F has to be mentioned explicitly.

3. SPECIFICATION LOGICS WITH CONSTRAINTS

In this section we review the concept of constraints for a specification logic SL. According to the approach in [Bal 90] we generalize the concept of constraints for algebraic specifications in [Eh 89] and [EM 90] including initial, generating, free generating and first order logical constraints leading to an induced specification logic SLC with constraints. We will show how pushouts, amalgamation and extension can be extended from the specification logic SL to SLC. These results will be applied in section 6 to behavioural module specifications with constraints.

3.1 DEFINITION (Logic of Constraints)

A logic of constraints $LC = (\text{Constr}, \models)$ on a given specification logic $SL = (\text{ASPEC}, \text{Catmod})$ (see 2.1) is given by a functor

$$\text{Constr}: \text{ASPEC} \rightarrow \text{Classes}$$

defined on the category ASPEC of abstract specifications with values in the category Classes of classes (see [HS 73] and [EM 90]) and for each object ASPEC in ASPEC a relation,

$$\models \subseteq \text{Obj}(\text{Catmod}(\text{ASPEC})) \times \text{Constr}(\text{ASPEC}),$$

called satisfaction relation for constraints, such that for all morphism $f: \text{ASPEC1} \rightarrow \text{ASPEC2}$ in ASPEC, all objects A_2 in $\text{Catmod}(\text{ASPEC2})$ and $C_1 \in \text{Constr}(\text{ASPEC1})$ we have the following satisfaction condition

$$A_2 \models f\#(C_1) \Leftrightarrow V_f(A_2) \models C_1$$

with $f\# = \text{Constr}(f)$ and $V_f = \text{Catmod}(f): \text{Catmod}(\text{ASPEC2}) \rightarrow \text{Catmod}(\text{ASPEC1})$.

Remark and Examples

1. The pair $\text{QINST} = (\text{SL}, \text{LC})$ can be considered as a quasi-institution because it corresponds exactly to the notion of an institution $\text{INST} = (\text{SIGN}, \text{Sen}, \text{Mod}, \models)$ in the sense of [GB 84] with $\text{SIG} = \text{ASPEC}$, $\text{Sen} = \text{Constr}$, $\text{Mod} = \text{Catmod}$, and the same kind of satisfaction condition \models , except of the fact that the target category of Sen is the category Sets of sets while that of Constr is the "category" Classes of classes (see [EM 90]).
2. Typical examples of constraints are initial, generating, free generating and first order logical constraints

which can be defined on the equational specification logic EQSL see [ER 89] and [EM 90]. Note, that initial and free generating constraints can be extended to an arbitrary specification logic SL even if SL does not have initial objects and free constructions in general, because the corresponding constraints require initial objects and free constructions only for specific abstract specifications resp. specific specification morphisms.

3.2 REMARK (Abstract Specifications with Constraints)

Given a logic of constraints $LC = (\text{Constr}, \models)$ on a specification logic $SL = (\text{ASPEC}, \text{Catmod})$ we are able to define abstract specifications with constraints as pairs $\text{ASPECC} = (\text{ASPEC}, C)$ consisting of an object ASPEC in ASPEC and a set of constraints $C \subseteq \text{Constr}(\text{ASPEC})$ and consistent specification morphisms $f: (\text{ASPEC1}, C1) \rightarrow (\text{ASPEC2}, C2)$. A specification formalism is consistent if we have $A2 \models C2 \Rightarrow A2 \models f\#(C1)$ for all ASPEC2 models A2. This leads to the category ASPECC of abstract specifications with constraints. Moreover, $V_f = \text{Catmod}(f): \text{Catmod}(\text{ASPEC2}) \rightarrow \text{Catmod}(\text{ASPEC1})$ can be restricted to a functor

$$VC_f = \text{Catmod}C(f): \text{Catmod}(\text{ASPEC2}, C2) \rightarrow \text{Catmod}(\text{ASPEC1}, C1)$$

where $\text{Catmod}C(\text{ASPEC}, C)$ is the full subcategory of $\text{Catmod}(\text{ASPEC})$ of all objects A satisfying C, i.e. $A \models C$. $\text{Catmod}: \text{ASPEC} \rightarrow \text{CATCAT}$ can thus be restricted to a functor $\text{Catmod}C: \text{ASPECC} \rightarrow \text{CATCAT}$.

3.3 DEFINITION AND FACT (Induced Specification Logic with Constraints)

Given a specification logic $SL = (\text{ASPEC}, \text{Catmod})$ and a logic of constraints $LC = (\text{Constr}, \models)$ on SL the pair

$$\text{SLC} = (\text{ASPECC}, \text{Catmod}C)$$

defined by (see 3.2)

$$\begin{aligned} \text{ASPECC} &= \text{Category of abstract specification with constraints, and} \\ \text{Catmod}C: \text{ASPECC} &\rightarrow \text{CATCAT} \end{aligned}$$

is a specification logic, called induced specification logic with constraints.

Remarks and Examples

1. Note that specifications built over the quasi-institution $\text{QINST} = (SL, LC)$ see remark of 3.1) correspond to abstract specifications with constraints ASPECC, which are now the "syntactical objects" in our induced specification logic SLC.
2. For each logic of constraints LC defined over the specification logic EQSL, CEQSL, BEQSL and VIEWSL (see 2.2) we obtain an induced specification logic EQSLC, CEQSLC, BEQSLC and VIEWSLC respectively (see 2.2.4).
3. On the other hand we can also consider (conditional) equations as constraints over a signature specification logic SIGSL such that $\text{SIGSLC} = \text{EQSL}$ (resp. $\text{SIGSLC} = \text{CEQSLC}$).

3.4 REMARK (Free Constructions in SLC)

1. Existence of free constructions in SL does not imply the same for SLC in general. For $f: \text{ASPEC1} \rightarrow \text{ASPEC2}$ in ASPEC with free construction $F_f: \text{Catmod}(\text{ASPEC1}) \rightarrow \text{Catmod}(\text{ASPEC2})$ and constraints $C1, C2$ s.t. we have a consistent specification morphism $f: (\text{ASPEC1}, C1) \rightarrow (\text{ASPEC2}, C2)$, we will not

automatically have a free construction $FC_f: \text{Catmod}(\text{ASPECC1}) \rightarrow \text{Catmod}(\text{ASPECC2})$ because $\text{Catmod}(\text{ASPECC2})$ may even be empty.

2. Sufficient for FC_f to exist is that F_f is persistent on C1, ie

$$A1 \models C1 \Rightarrow V_f(F_f(A1)) = A1,$$

and that in C2 we have the set of induced constraints on our hand, ie

$$C2 = f\#(C1) \cup \text{FGEN}(f),$$

where $\text{FGFN}(f)$ is the free generating constraint w.r.t. f. It is satisfied by a model $A2$ if we have $A2 = F_f(V_f(A2))$. Given these conditions F_f can be restricted to a strongly persistent free functor FC_f . In other words the morphism $f: (\text{ASPEC1}, C1) \rightarrow (\text{ASPEC2}, C2)$ is strongly liberal in this case (note that the definition of the induced constraints guarantees f to be consistent).

3. Free constructions in an induced specification logic SLC over a signature specification logic SL are often obtained not as restrictions of the free constructions in SL but via a construction depending on the kind of the constraints in SLC. In EQSL, for example, free images along a specification morphism are build via a free construction in the first step and a factorization afterwards (see [EM 85] for details).

3.5 THEOREM (Pushouts in SLC)

The induced specification logic SLC has pushouts iff SL has pushouts.

Proofidea

For $f_i: (\text{ASPEC0}, C0) \rightarrow (\text{ASPECi}, Ci)$ in ASPECC and $i = 1, 2$ the pushout object $(\text{ASPEC3}, C3)$ of $f1$ and $f2$ in ASPECC is defined by the corresponding pushout ASPEC3 of $f1$ and $f2$ in ASPEC with morphisms $g_i: \text{ASPECi} \rightarrow \text{ASPEC3}$ for $i = 1, 2$ and $C3 = g1\#(C1) \cup g2\#(C2)$ (see [Bal 90], [EM 90]).

3.6 THEOREM (Amalgamation in SLC)

The induced specification logic SL has amalgamations iff SL has amalgamations.

Proofidea

Given A_i in $\text{Catmod}(\text{ASPECCi})$ for $i = 0, 1, 2$ with $VC_{f1}(A1) = A0$ and $VC_{f2}(A2) = A0$ then we also have $V_{f1}(A1) = A0$ and $V_{f2}(A2) = A0$ such that the amalgamation $A3 = A1 +_{A0} A2$ in SL is defined which becomes also an amalgamation in SLC because by the satisfaction condition $A3$ satisfies $C3$ as defined in the proofidea of 3.5 (see [Bal 90], [EM 90]).

3.7 COROLLARY (Extension in SLC)

The induced specification logic SLC has extensions if SL has amalgamations.

Proof

Direct consequence of 2.7 and 3.6.

3.8 REMARK (Restricted Extension in SLC)

If SL does not have amalgamation in general but only extensions we would like to conclude extensions for SLC. Unfortunately we cannot conclude this in general but only for all those strongly persistent functors $FC: \text{Catmod}C(\text{ASPEC0}, C0) \rightarrow \text{Catmod}C(\text{ASPEC1}, C1)$ which are already restrictions of strongly persistent functors $F: \text{Catmod}C(\text{ASPEC0}) \rightarrow \text{Catmod}C(\text{ASPEC1})$. In order to conclude part 2 of 2.6 for SLC we have to assume that the strongly persistent free functor FC is the restriction of a strongly persistent free functor F.

Proofidea

Give FC as restriction of a strongly persistent functor F the assumption that SL has extensions implies that we have a strongly persistent extension F^* of F. The restriction of F^* to objects in $\text{Catmod}C(\text{ASPEC2}, C2)$ leads to a strongly persistent extension FC^* of FC. Moreover, FC^* is a strongly persistent free functor, provided that FC is strongly persistent free and the restriction of a free functor F. This implies that F^* is free because SL has extensions and hence also the restriction FC^* of F^* is free.

3.9 REMARK (Signature Specification Logics)

As pointed out in example 3 of 3.3 several specification logics like EQSL and CEQSL can be considered as induced specification logics over certain signature specification logic like SIGSL. This means that in order to show pushouts and amalgamation in EQSL resp. CEQSL it suffices to show pushouts and amalgamation in the corresponding signature specification logic SIGSL. In cases like VIEWSL, however, where a specification logic is not induced by a signature specification logic, the existence of pushouts and amalgamations cannot be established as a consequence of abstract properties (see 2.2.4 and section 5).

4. ABSTRACT MODULE SPECIFICATIONS

The algebraic module specification concept was developed in joint cooperation of researchers in theoretical computer science and software engineering (see [EW 85], [EW 86], [WE 86], [BEP 87], and [EM 89]). The main idea was to extend the concept of parameterized specifications (see [TWW 78/82] and [EM 85]) by explicit import and export interfaces and to provide "horizontal operations" on module specifications, like composition, union and actualization, which allow to build up specifications for interconnected systems from basic units. In this section we present the module specification concept together with the composition operation and the main correctness and compositionality result on the level of abstract specifications as introduced in section 2 and discuss how the remaining theory of algebraic module specifications presented in [EM 90] chapters 2 and 3 can also be extended.

4.1 General Assumptions and Remarks

We assume to have a specification logic $SL = (\text{ASPEC}, \text{Catmod})$ which has pushouts, free constructions, and extensions with respect to all or at least special pushout situations in ASPEC. All of this assumptions are satisfied in the most general way by the specification logic EQSL of equational algebraic specifications SPEC (see 2.2.1) where $\text{Cat}(\text{SPEC})$ is the category of all SPEC-algebras and SPEC-homomorphisms. This leads to the concept of module specifications as considered in chapters 1 - 6 of [EM 89]. On first reading one should consider only this special case which means that all "abstract algebraic specifications" ASPEC can be replaced by "equational algebraic specifications" SPEC and similar for morphisms and models. In special cases free constructions and extensions are available also in the specification logics BEQSL, VIEWSL, BEQSLC and VIEWSLC to be considered in sections 5 and 6.

4.2 DEFINITION (Abstract Module Specification)

1. An abstract module specification

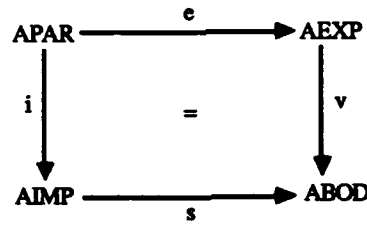
$$\text{AMOD} = (\text{APAR}, \text{AEXP}, \text{AIMP}, \text{ABOD}, e, s, i, v)$$

consists of abstract algebraic specifications

- APAR (abstract parameter specification)
- AEXP (abstract export interface specification)
- AIMP (abstract import interface specification)
- ABOD (abstract body specification)

and abstract specification morphisms e, s, i, v such that the following diagram commutes, i.e.

$$v \circ e = s \circ i:$$



2. The functorial semantics SEM of AMOD is the functor

$$\text{SEM} = V_v \circ F_s: \text{Cat}(\text{AIMP}) \rightarrow \text{Cat}(\text{AEXP})$$

where $F_s: \text{Cat}(\text{AIMP}) \rightarrow \text{Cat}(\text{ABOD})$ and $V_v: \text{Cat}(\text{ABOD}) \rightarrow \text{Cat}(\text{AEXP})$ are the free and forgetful functors corresponding to s and v respectively. The semantics SEM is defined only if the free functor exists.

3. The abstract module specification AMOD is called (internally) correct if the free functor F_s exists and is strongly persistent, i.e. $V_v \circ F_s = \text{ID}_{\text{Cat}(\text{AIMP})}$ where s is a strongly liberal morphism.

Remarks and Interpretation

Module specifications include parameterized specifications with initial semantics (see [EM 85] chapters 7 and 8) as a special case where i and v are identities and $e = s$.

The semantics SEM of AMOD can be considered as a transformation from import data types in $\text{Cat}(\text{AIMP})$ to export data types in $\text{Cat}(\text{AEXP})$. Internal correctness of AMOD means that each import data type I is protected within the free construction F_s , because we have $V_v \circ F_s(I) = I$. For other notions of correctness see [EM 90] chapter 2.

4.3 DEFINITION (Composition of Abstract Module Specifications)

Given abstract module specifications

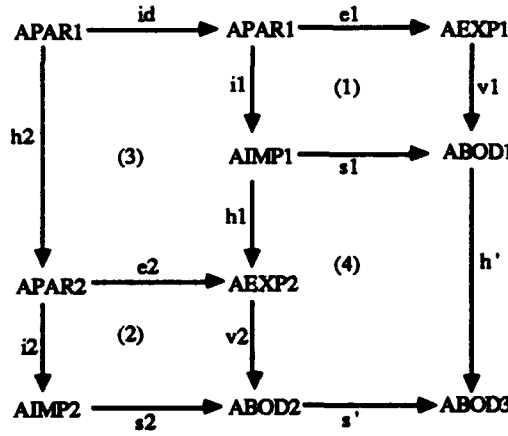
$$\text{AMOD}_j = (\text{APAR}_j, \text{AEXP}_j, \text{AIMP}_j, \text{ABOD}_j, e_j, i_j, s_j, v_j) \quad \text{for } j = 1, 2$$

and a pair $h = (h_1, h_2)$ of abstract specification morphisms satisfying $e_2 \circ h_2 = h_1 \circ i_1$ the composition of AMOD1 and AMOD2 via h is an abstract module specification

$$\text{AMOD}_3 = \text{AMOD}_1 \circ_h \text{AMOD}_2$$

defined by the outer square in the following diagram with $\text{AEXP}_3 = \text{AEXP}_1$, $\text{AIMP}_3 = \text{AIMP}_2$, $\text{APAR}_3 =$

APAR1 and ABOD3 constructed as pushout object in subdiagram (4).



Remarks and Interpretation

The specification morphisms $h1: AIMP2 \rightarrow AEXP2$ and $h2: APAR1 \rightarrow APAR2$ match the import interface AIMP1 of AMOD1 with the export interface AEXP2 of AMOD2 and APAR1 with APAR2. The remaining import interface AIMP2 becomes the import of AMOD3 and the remaining export interface AEXP1 becomes the export of AMOD3. The new body ABOD3 can be considered as union of ABOD1 and ABOD2 with shared subpart AIMP1. The abstract specification morphisms of AMOD3 are explicitly given by: $e3 = e1$, $i3 = i2 \circ h2$, $s3 = s' \circ s2$, and $v3 = h' \circ v1$ where s' and h' are defined by the pushout (4).

4.4 THEOREM (Correctness and Compositionality of Composition)

Given correct abstract module specifications AMOD1 and AMOD2 and h as in 4.3 such that the composition $AMOD3 = AMOD1 \circ_h AMOD2$ is defined then we have

- (a) AMOD3 is correct (correctness), and
- (b) $SEM3 = SEM1 \circ V_{h1} \circ SEM2$ (compositionality)

for the functorial semantics SEM_j of AMOD $_j$ for $j = 1, 2, 3$ provided that the specification logic SL has extensions for the pushout (4) in 4.3.

Proof

Since AMOD1 and AMOD2 are correct we have strongly persistent free functors $FREE_{s1}$ and $FREE_{s2}$. Since SL has extensions for the pushout (4) we have

- (c) strong persistency of $FREE_{s'}$, and
- (d) $FREE_{s1} \circ V_{v2} \circ h1 = V_{h'} \circ FREE_{s'}$.

From (c) we conclude strong persistency of $FREE_{s3} = FREE_{s'} \circ FREE_{s2}$. $FREE_{s3}$ is free construction w.r.t. V_{s3} because free constructions are closed under composition. Hence we have correctness of AMOD3 (a) and (d) implies compositionality (b):

$$\begin{aligned}
 SEM3 &= V_{v3} \circ FREE_{s3} && \text{(def of SEM 3)} \\
 &= V_{v1} \circ V_{h'} \circ FREE_{s'} \circ FREE_{s2} && \text{(def of } h3 \text{ and } s3) \\
 &= V_{v1} \circ FREE_{s1} \circ V_{h1} \circ V_{v2} \circ FREE_{s2} && \text{(condition (c) above)} \\
 &= SEM1 \circ V_{h1} \circ SEM2 && \text{(def of SEM1 and SEM2).}
 \end{aligned}$$

Similar to composition we can also generalize union and actualization as presented in [BEP 87] and [EM 90] from equational algebraic specifications to an arbitrary specification logic with pushouts, free constructions, amalgamation and extension. Note, that for union and actualization we use amalgamation (which is not necessary for composition but useful to imply extension as shown in 2.7) in order to formulate and prove that the semantics is compositional. Moreover, all of the compatibility results between these operations which have been expressed by associativity, commutativity, and distributivity laws using pushout and colimit techniques in [BEP 87] and [EM 90] can be extended to each specification logic with pushouts.

4.5 DISCUSSION (Module Specifications with Induced Body Constraints and Vertical Refinement)

From remark 3.4 (free construction in SLC) we conclude that module specifications over an induced specification logic with constraints SLC are not guaranteed to carry any semantics at all - be it correct or not - even if in the underlying specification logic SL every specification morphism is liberal. To tackle this problem one might choose to consider module specifications with an empty set of constraints in the body specification. Such a module

$$\text{AMODC} = ((\text{APAR}, \text{CP}), (\text{AEXP}, \text{CE}), (\text{AIMP}, \text{CI}), \text{ABOD}, e, s, i, v)$$

consists out of an underlying module

$$\text{AMOD} = (\text{APAR}, \text{AEXP}, \text{AIMP}, \text{ABOD}, e, s, i, v)$$

over SL and sets of constraints CP, CE and CI which have to be subsets of $\text{Constr}(\text{APAR})$, $\text{Constr}(\text{AEXP})$ resp. $\text{Constr}(\text{AIMP})$. Provided that SL has free constructions we attach a reasonable semantics to each AMODC simply by restricting the semantics $\text{SEM} = V_v \circ F_s : \text{Cat}(\text{AIMP}) \rightarrow \text{Cat}(\text{AEXP})$ of AMOD to $\text{Cat}((\text{AIMP}, \text{CI}))$. Pragmatically this is justified if we think of constraints as of interest only in parameter, export and import while the body is an invisible constructive description where arbitrary constraints have no role to play.

To obtain a notion of correctness the set of induced body constraints

$$\text{CB} = s\#(\text{CI}) \cup \text{FGEN}(s)$$

is introduced (see remark 3.4). A module AMODC is said to be correct if the module

$$((\text{APAR}, \text{CP}), (\text{AEXP}, \text{CE}), (\text{AIMP}, \text{CI}), (\text{ABOD}, \text{CB}), e, s, i, v)$$

is a correct module over SLC in the sense of definition 4.2.3. This means that the morphisms are morphisms in SLC and that $s : (\text{APAR}, \text{CP}) \rightarrow (\text{ABOD}, \text{CB})$ is strongly liberal in SLC. Correctness of AMODC follows if SEM is constraints preserving, i.e.

$$A \models \text{CI} \Rightarrow \text{SEM}(A) \models \text{CE},$$

F_s is persistent on CI, $\text{CE} \Rightarrow e\#(\text{CP})$ and $\text{CP} \Rightarrow i\#(\text{CI})$ (see [Bal 90] and [EM 90]). Under these assumptions the notion of induced constraints from remark 3.4 is employed to tell us that s is strongly liberal in SLC with respect to CB.

In [EM 90] only induced constraints are allowed in the bodies of parameterized specifications with constraints as well as of modular specifications with constraints. In [Bal 90] this approach is generalized from EQSLC to arbitrary specification logics with constraints. In this paper, however, we allow arbitrary constraints in module bodies resp. target specifications and require morphisms specifically to be (strongly) liberal.

Concerning vertical refinement of module specifications as discussed in chapters 5 and 6 of [EM 90] there seems to be no big problem in formulating basic parts of the theory in the framework of specification logics.

5. BEHAVIOURAL-MODULE-SPECIFICATIONS

In this section we present the behavioural equational specification logic BEQSL and based upon it the theory of behavioural-module-specifications together with the basic notions needed in this context. This approach is inspired by the common programming technique to implement an abstract datatype by a structure which satisfies the desired properties only wrt. to its external behaviour. For example a standard implementation of STACK(DATA) using an array with a top pointer will not satisfy $\text{POP}(\text{PUSH}(X,S)) = S$ if the top element is not deleted when POP is performed. Such an implementation, however, is a behaviour model of a suitable behavior specification where the STACK sort is declared non observable.

The theory as presented here is completely based upon the theory of equational algebraic specifications introduced in [EM 85], [EM 90], and behavioral semantics in the sense of [NO 88] and [ONE 89].

5.1 BASIC NOTIONS

A behavioural specification $\text{BSP} = (\text{Obs}, S, \text{OP}, E)$ is an equational specification $\text{SP} = (S, \text{OP}, E)$ together with a subset $\text{Obs} \subseteq S$ of observable sorts. The sorts $\text{NoObs} := S - \text{Obs}$ are called non-observable sorts.

Given two behaviour-specifications $\text{BSP}_i, i = 1, 2$ a behaviour-specification-morphism $h: \text{BSP}_1 \rightarrow \text{BSP}_2$ is a specification-morphism $h: \text{SP}_1 \rightarrow \text{SP}_2$ preserving both the observable and the non-observable part, i.e. $h_S(\text{Obs}_1) \subseteq \text{Obs}_2$ and $h_S(\text{NoObs}_1) \subseteq \text{NoObs}_2$.

A specification morphism preserving only the non-observable part is called view-specification-morphism.

BSPEC is the category of all behaviour-specifications together with the corresponding behaviour-specification-morphisms.

View-BSPEC is the category of all behaviour-specifications together with the corresponding view-specification-morphisms.

Given a behaviour-specification BSP and a possibly non-observable sort s in BSP a BSP-context over s is a term $c[z] \in \text{TOP}(X_{\text{Obs}} \cup \{z\})$ of observable sort with z being a variable of sort s .

Given an arbitrary term t of sort s the application of $c[z]$ over t is the term resulting from the substitution of z by t in $c[z]$.

An SP-algebra A is a BSP-algebra if it satisfies behaviorally each of the equations e in E .

An algebra A satisfies an equation $e: \lambda Y \cdot t_1 = t_2$ behaviorally if it satisfies all of its observable consequences.

An observable consequence of e is given for any BSP-context $c[z]$ over the sort of e for each assignment $\sigma: Y \rightarrow \text{TOP}(X_{\text{Obs}})$ eliminating the variables of non-observable sorts in e by $e': \lambda X_{\text{Obs}} \cdot c[\sigma^*(t_1)] = c[\sigma^*(t_2)]$.

Given a behaviour-specification BSP an observable BSP-computation w.r.t. a family of sets of variables of observable sort is a term $t \in \text{TOP}(X_{\text{Obs}})_{\text{Obs}}$, i.e. a term of observable sorts containing only observable variables.

Analogously given a BSP-algebra A an observable computation t over A is an OP-term of observable sort built upon the A_{Obs} -elements as constants, i.e. $t \in \text{TOP}(A_{\text{Obs}})_{\text{Obs}}$.

Given two BSP-algebras A, B a BSP-behaviour-morphism $f: A \rightarrow B$ (analogous to the notion of homomorphisms) is a family $f = (f_s)_s \in \text{Obs}$ of mappings preserving all observable computations $t \in \text{TOP}(A_{\text{Obs}})_{\text{Obs}}$.

Beh(BSP) is the model-category w.r.t. a behaviour-specification BSP consisting of all BSP-algebras together with the corresponding BSP-behaviour-morphisms.

The forgetful functor $\text{BU}_h: \text{Beh}(\text{BSP}_2) \rightarrow \text{Beh}(\text{BSP}_1)$ w.r.t. a behaviour-specification-morphism $h: \text{BSP}_1 \rightarrow \text{BSP}_2$ is defined as in the non-behavioural case.

If h is a view-specification-morphism a special view-functor $\text{View}_h: \text{Beh}(\text{BSP}_2) \rightarrow \text{Beh}(\text{BSP}_1)$ has to be defined: for every algebra $A_2 \in \text{Beh}(\text{BSP}_2)$ $\text{View}_h(A_2)$ is defined to be the image of the ordinary forgetful functor applied to the algebra $\text{TOP}_2(A_2_{\text{Obs}_2}) \models_{h(E_1) \cup \text{eobs}(A_2)} \text{TOP}_2(A_2_{\text{Obs}_2})$ denotes the term-algebra

of the BSP_2 -signature built upon the $A2_{Obs2}$ -constants and $eobs(A_2)$ is the set of all observable equations satisfied by A_2 .

For more details see [NO 88] or [Cor 90].

5.2 DEFINITION AND FACT (The Behavioural Equational Specification Logic BEQSL)

1. Let $BCatmod:BSPEC^{OP} \rightarrow CATCAT$ be the functor defined as

- $BCatmod(BSP) := Beh(BSP)$ for any behaviour-specification BSP
- $BCatmod(h) := BU_h$ for any behaviour-specification-morphism h

The behavioural equational specification logic is defined as the tuple $BEQSL = (BSPEC, BCatmod)$.

2. Behaviour specification together with behaviour morphisms and behavior satisfaction is not an institution. To see this consider

$$\begin{array}{lll} BSIG1 = & & BSIG2 = BSIG1 + \\ \text{sorts } s1 \text{ nonobs} & \text{and} & \text{sorts } s2 \text{ obs} \\ \text{opns } a, b: \rightarrow s1 & & \text{opns } f: s1 \rightarrow s2 \end{array}$$

with the obvious inclusion morphism and the $BSIG2$ -algebra $A2 = (\{1,2\}, \{1,2\}, 1,2, id)$. The forgetful $BSIG1$ -image $A1 = (\{1,2\}, 1,2)$ satisfies the equation $a = b$ behaviourally because the set of observable consequences wrt. $BSIG1$ is empty. $A2$ itself, however, does not satisfy the translated equation (again $a = b$) because $1 \neq 2$ in $A2$ violates the observable consequence $f(a) = f(b)$ which arises through the context $f(z)$. This means that the satisfaction condition is not fulfilled. The same argument applies if we consider view morphisms instead of behaviour morphisms because the inclusion morphism in our counter example is also a view morphism.

5.3 THEOREM (Properties of BEQSL)

The behavioural equational specification logic $BEQSL$ has free constructions, pushouts and, for a specific class of pushouts, amalgamations and extensions:

Proof Idea

1. Let $h: BSP_1 \rightarrow BSP_2$ be a behaviour-specification-morphism and $A1 \in Beh(BSP_1)$ an arbitrary BSP_1 -algebra. The algebra defined by: $BFree_h(A1) = TOP_2(A1_{Obs1}) \models_{h(eobs(A1))} E2$ is freely generated over $A1$ w.r.t. the forgetful functor BU_h . This construction is very similar to that of the view functor, i.e. a special kind of quotient-term algebra built upon the observable parts of $A1$ taken as constants of sort $h(s)$ for every $s \in Obs1$.

2. Given behaviour-specifications $BSP_i, i = 0, 1, 2$ and behaviour-specification-morphisms $i1: BSP_0 \rightarrow BSP_1, h1: BSP_0 \rightarrow BSP_2$. Let SP_3 denote the pushout of SP_1 and SP_2 in the category $SPEC$, $i2: SP_2 \rightarrow SP_3$ and $h2: SP_1 \rightarrow SP_3$ the corresponding morphisms.

Then $BSP_3 := (h2^\#(Obs1) + i2^\#(Obs2), SP_3)$ is a pushout in the category $BSPEC$, $i2$ and $h2$ interpreted as behaviour-specification-morphisms. Let us mention that the category $View-BSPEC$ has pushouts as well. The construction is similar to the one above, with the difference, that the non-observable sorts $NoObs_3$ are determined by the pushout-sum $NoObs_3 = h2^\#(NoObs_1) + i2^\#(NoObs_2)$.

3. The behavioural equational specification logic $BEQSL$ has amalgamations w.r.t. all $BSPEC$ -pushouts satisfying the so-called observation preserving property. More details concerning this property are found in [NO

88] and especially in [Cor 90].

The construction of the amalgamated algebras is identical to the non-behaviour case; amalgamated behaviour-morphisms are constructed similarly but only for the observable part.

Behaviour-amalgamation has the same uniqueness properties as the usual amalgamation.

4. BEQSL has extensions via amalgamations for the same class of pushouts satisfying the observation preserving property, so this part is a consequence of theorem 2.7.

The following definitions 5.4, 5.5 and theorem 5.6 are examples for the abstract case of definitions 4.2, 4.3 and theorem 4.4, taking into account the notions and special problems of the behaviour theory.

5.4 DEFINITION (Behaviour-Module-Specifications)

1. A behaviour-module-specification $\text{BMOD} = (\text{BPAR}, \text{BEXP}, \text{BIMP}, \text{BBOD}, e, s, i, v)$ consists of four behaviour-specifications and four behaviour-specification-morphisms e, s, i, v satisfying: $v \circ e = s \circ i$.
2. The functional behaviour-semantics BSEM of BMOD is the functor $\text{BSEM} = \text{BU}_v \circ \text{BFree}_s: \text{Beh}(\text{BIMP}) \rightarrow \text{Beh}(\text{BEXP})$ where $\text{BFree}_s: \text{Beh}(\text{BIMP}) \rightarrow \text{Beh}(\text{BBOD})$ and $\text{BU}_v: \text{Beh}(\text{BBOD}) \rightarrow \text{Beh}(\text{BEXP})$ are the behaviour-free and forgetful functors corresponding to s and v , respectively. BSEM always exists because BEQSL has free constructions.
3. BMOD is called (internally) correct if the free functor BFree_s is strongly persistent, i.e. $\text{BU}_s \circ \text{BFree}_s = \text{ID}_{\text{Beh}(\text{BIMP})}$.

Remark

BMOD is a module-specification based upon the specification-logic BEQSL. For this reason the morphisms e, i, s, v have to be behaviour-specification-morphisms. In some practical applications, however, it seems reasonable to allow v to be a view-specification-morphism, see for example in section 7 the result of the view-composition. This leads to a so-called view-module, introduced in [Cor 90]. Concerning the general problems around view-specification-morphisms see 5.7.

5.5 DEFINITION (Behaviour-Composition)

Given behaviour-module-specifications $\text{BMOD}_1, \text{BMOD}_2$ and a behaviour-composition-morphism $h = (h1, h2)$ consisting of behaviour-specification-morphisms $h1: \text{BIMP}_1 \rightarrow \text{BEXP}_2, h2: \text{BPAR}_1 \rightarrow \text{BPAR}_2$ satisfying: $h1 \circ i1 = e2 \circ h2$ the behaviour-composition of BMOD_1 and BMOD_2 via h is a behaviour-module-specification $\text{BMOD}_3 = \text{BMOD}_1 \circ_h \text{BMOD}_2$ defined by $\text{BEXP}_3 := \text{BEXP}_1, \text{BIMP}_3 := \text{BIMP}_2, \text{BPAR}_3 := \text{BPAR}_1$ and BBOD_3 constructed as behaviour-pushout of $s1$ and $v2 \circ h1$ analogous to the construction in 4.3. Behaviour-composition is called behaviour-consistent, if the construction pushout satisfies the observation-preserving-property.

5.6 THEOREM (Correctness and Compositionality of Behaviour-Composition)

Given correct behaviour-module-specifications BMOD_1 and BMOD_2 and a behaviour-composition-morphism h such that the composition $\text{BMOD}_3 = \text{BMOD}_1 \circ \text{BMOD}_2$ is defined and behaviour-consistent, then we have

- $\text{BSEM}_3 = \text{BSEM}_1 \circ \text{BU}_{h1} \circ \text{BSEM}_2$ for the functorial behaviour-semantics $\text{BSEM}_i, i = 1, 2, 3$ (compositionality)

- BMOD_3 is (internally) correct (correctness)

Proof

The observation preserving property of the construction pushout guarantees extensions via amalgamations. So this theorem is a consequence of the abstract case 4.4.

Remark

Not only composition but also union and actualization show the same nice properties concerning correctness and compositionality. In any case the observation preserving property has to be required for the corresponding pushout-constructions defining the behaviour-consistency of these operations.

5.7 DISCUSSION (The View Case)

The view case is characterized by allowing the more general view-specification-morphisms instead of the restrictive behaviour-specification-morphisms.

The necessity for using this kind of morphisms becomes obvious if, for example, the data-part in a parameterized specification $\text{set}(\text{data})$ is actualized by $\text{set}(\text{nat})$ leading to sets of sets of natural numbers. Being data observable and set non-observable, data has to be assigned to the sort set and this is impossible if we are confined to behaviour-specification-morphisms.

If we consider the view-equational-specification logic $\text{VIEWSL} = (\text{View-BSPEC}, \text{VCatmod})$ defined by $\text{VCatmod}(\text{BSP}) = \text{Beh}(\text{BSP})$ for any behaviour-specification BSP, and $\text{VCatmod}(h) = \text{View}_h$ for any view-specification-morphism h , we obtain a rather "poor" specification-logic only having pushouts, but in general no longer free constructions nor amalgamations nor extensions.

Every behaviour-specification-morphism is a view-specification-morphism as well, thus we have those properties in special cases.

The most general "special case" satisfying at least a restricted form of the extension lemma is a pushout-diagram in which only the vertical morphisms are real view-specification-morphisms.

Then for the horizontal part we have free constructions, and if a special observation preserving property is satisfied we have a special extension, but no amalgamation. For more details see [NO 88], [Cor 90] and for a very restricted "amalgamation in the free case" [ON 90].

A problem left is the notion of persistency. Usually strong persistency of a functor F w.r.t. V is defined like in definition 2.3 of this paper: $V \circ F = \text{ID}$.

Applying this notion in the behaviour theory we have no longer that for any persistent functor F there is a naturally isomorphic strongly persistent functor F' : $F \simeq F'$.

This problem should be solved by introducing a more appropriate notion of strong persistency operating with a functor "B-ID" requiring syntactical identity only on the observable parts.

In the framework of behaviour-module-specifications it seems to be useful to allow v to be a view-specification-morphism. So a non-observable body-sort could be defined observable in the export-part. One could then use view morphisms as passing morphisms because in general this induces v in the resulting module to be a view morphism. See example 7 for an application where v , however, remains a behaviour morphism.

Also for parameter-passing-morphisms in behaviour-actualization view-specification-morphisms are appropriate in practical applications.

6. BEHAVIOURAL-MODULE-SPECIFICATIONS WITH CONSTRAINTS

In this section we generalize the results of section 5 to the case of behaviour-specifications with arbitrary constraints. In fact, we apply the general theory of specification logics with constraints in section 3, and the general theory of module specifications in section 4, to the case of behavioral specifications with constraints.

6.1 DEFINITION (Behaviour-Specification with Constraints)

Given a logic of behaviour-constraints $BLC = (BConstr, \models)$ (see definition 3.1) with $BConstr: BSPEC \rightarrow$ Classes a behaviour-specification with constraints $BSPC$ w.r.t. BLC is a pair $BSPC = (BSP, BC)$ with BSP being a behaviour-specification and $BC \subseteq BConstr(BSP)$ a set of constraints on BSP .

A consistent behaviour-specification-morphism $h: BSPC_1 \rightarrow BSPC_2$ is defined as a behaviour-specification-morphism $h: BSP_1 \rightarrow BSP_2$ satisfying for any algebra $A_2 \in Beh(BSP_2)$: $(A_2 \models C_2) \Rightarrow (A_2 \models h\#(C_1))$

The category $BSPECC$ consists of all behaviour-specifications with constraints together with the corresponding consistent behaviour-specification-morphisms.

6.2 DEFINITION (BEQSLC)

The behavioural equational specification logic with constraints $BEQSLC = (BSPECC, BCatmodC)$ is the induced specification logic (see 3.3) by the specifications logic $BEQSL$ (see 5.2) and the logic of behaviour constraints BLC (see 6.1).

Remark

A typical example of a logic of behaviour-constraints in addition to those mentioned in remark 2 of 3.1 is the logic of view-free-generating constraints:

(b, v) is called view-free-generating constraint on BSP if $b: BSP_0 \rightarrow BSP_1$ is a behaviour-specification-morphism and $v: BSP_1 \rightarrow BSP$ is a view-specification-morphism.

A BSP -algebra satisfies (b, v) iff:

$$View_v(A) \equiv_B BFree_b \circ BU_b \circ View_v(A).$$

The behaviour-equivalence " \equiv_B " means that there is a BSP_1 -behaviour-isomorphism.

The translation of constraints is defined for $h: BSP \rightarrow BSP'$ by $h\#((b, v)) := (b, h \circ v)$.

View-free-generating constraints do not generalize syntactically behaviour-free-generating constraints because being v a behaviour-specification-morphism the algebras $View_v(A)$ and $BU_v(A)$ are only behaviour-equivalent but not identical.

6.3 COROLLARY (Pushouts in BEQSLC)

The specification-logic $BEQSLC$ has pushouts (see 3.5 and 5.3).

6.4 FACT (Amalgamation and Extension in BEQSLC)

Pushouts w.r.t. the specification-logic $BEQSLC$ satisfying the observation preserving property have amalgamations and extensions.

Proof

Amalgamation- and extension-lemma with constraints, respectively, are proven "pushout-wise" using the amalgamation and extension-properties of the underlying pushouts without constraints. Therefore the observation preserving property for pushouts with constraints has to be defined in the same way as for the corresponding pushouts without constraints. Then, the fact is a consequence of 3.6, 3.7, and 5.3.

6.5 DEFINITION (Behaviour Module Specifications with Constraints)

1. A behaviour-module-specification with constraints $\text{BMODC} = (\text{BPARC}, \text{BIMPC}, \text{BEXP}, \text{BBODC}, e, s, i, v)$ consists of four behaviour-specifications with constraints and four consistent behaviour-specifications-morphisms, such that $v \circ e = s \circ i$.
2. The functorial behaviour-semantics BSEMC of BMODC is the functor $\text{BSEMC} = \text{BU}_v \circ \text{CBFree}_s: \text{Beh}(\text{BIMPC}) \rightarrow \text{Beh}(\text{BEXP})$ where $\text{CBFree}_s: \text{Beh}(\text{BIMPC}) \rightarrow \text{Beh}(\text{BBODC})$ and $\text{BU}_v: \text{Beh}(\text{BBODC}) \rightarrow \text{Beh}(\text{BEXP})$ are the behaviour-free and forgetful functors corresponding to s and v , respectively.
The semantics BSEMC is defined if the behaviour-free functor CBFree_s exists.
3. BMODC is called (internally) correct, if the functorial behaviour-semantics BSEMC exists and CBFree_s is strongly persistent, i.e. $\text{BU}_s \circ \text{CBFree}_s = \text{ID}_{\text{Beh}(\text{BIMPC})}$.

Remarks

1. The free functor CBFree_s has not to be a restriction of the free functor $\text{BFree}_s: \text{Beh}(\text{BIMP}) \rightarrow \text{Beh}(\text{BBOD})$, i.e. it is not necessarily $\text{CBFree}_s = \text{BFree}_s \upharpoonright \text{Beh}(\text{BIMPC})$. We only require the existence of an arbitrary free functor.
2. The semantics - if existing - is constraints-preserving, i.e. being $A \in \text{Beh}(\text{BIMP})$:

$$A \models \text{C}_{\text{IMP}} \Rightarrow \text{BSEMC}(A) \models \text{C}_{\text{EXP}}.$$

This is a consequence of the satisfaction-condition: $B \models v\#(\text{C}_{\text{EXP}}) \Leftrightarrow \text{BU}_v(B) \models \text{C}_{\text{EXP}}$ for any BBODC -algebra B , and the left-hand side is always satisfied because v is consistent.

6.6 DEFINITION (Behaviour-Composition with Constraints)

Given behaviour-module-specifications with constraints BMODC_1 , BMODC_2 and a consistent behaviour-composition-morphism $h = (h1, h2)$, $h1: \text{BIMPC}_1 \rightarrow \text{BEXP}_2$, $h2: \text{BPARC}_1 \rightarrow \text{BPARC}_2$ satisfying $h1 \circ i1 = e2 \circ h2$, the behaviour-composition with constraints of BMODC_1 and BMODC_2 via h is a behaviour-module-specification with constraints $\text{BMODC}_3 = \text{BMODC}_1 \circ_h \text{BMODC}_2$ defined analogous to the case without constraints in 5.5. Behaviour-composition with constraints is called behaviour-consistent, if the construction pushout satisfies the observation preserving property with constraints.

6.7 THEOREM (Correctness and Compositionality of Behaviour-Composition with Constraints)

Given correct behaviour-module-specifications with constraints BMODC_1 , BMODC_2 and a consistent behaviour-composition-morphism h , such that the composition with constraints $\text{BMODC}_3 = \text{BMODC}_1 \circ_h \text{BMODC}_2$ is defined and behaviour-consistent, then we have:

- the functorial behaviour-semantics BSEMC_3 of BMODC_3 is defined and we have
 $\text{BSEMC}_3 = \text{BSEMC}_1 \text{ BU}_{h1} \circ \text{BSEMC}_2$ (compositionality)
- BMODC_3 is correct (correctness)

Proof

This is a consequence of our theorems 5.3 and 4.4. □

Finally let us mention (see [Cor 90]) that union and actualization with constraints can be defined in the same way such that we obtain correctness and compositionality-properties. This theorem is a consequence of the non-constraints case, 5.6, and the fact 6.4.

7. EXAMPLE (BEHAVIORAL CASE OF AIRPORT SCHEDULE MODULES)

The airport schedule system specified in [EM 90] can be studied under the behavioural approach, defining the sort aps as non-observable in the aps-Behaviour-module according to definition 5.4:

```

APS-BPAR = BOOL +
    sorts    flight-number    (flight#)    obs
            destination      (dest)       obs
            departure         (dep)        obs
            plane-number     (plane#)     obs
            type              obs
            seats             obs
    opns     F-EQ:flight# flight#    →    bool
            P-EQ:plane# plane#      →    bool
    NO-DEPART: → dep
    eqns     ...

APS-BEXP = APS-BPAR +
    sorts    airport-schedule (aps)        non-obs
    opns     CREATE: → aps
            SCHEDULE: → flight# dest dep plane type seats aps → aps
            SEARCH: flight# aps → bool
            RETURN: flight# aps → dep
            CHANGE: flight# dest aps → aps
    eqns     ...

APS-BIMP = APS-BPAR +
    sorts    flight-schedule (fs)          obs
            plane-schedule (ps)          obs
    opns     CREATE-FS: → fs
            CREATE-PS: → ps
            ADD-FS: flight# dest dep fs → fs
            RESERVE-PS: plane# type seats ps → ps
            SEARCH-FS: flight# fs → bool
            SEARCH-PS: plane# ps → bool
            RETURN-FS: flight# fs → dep
            CHANGE-FS: flight# dep fs → fs
    eqns     ..

APS-BBOD = (APS-BEXP + APS-BPAR APS-BIMP) +
    opsn     TUP: fs ps → aps
    eqns     ...

```

Airport schedules are constructed by means of a hidden function TUP in the body-part of the aps-module, joining (imported) flight and plane schedules, respectively. This function is hidden in the body because in general it constructs inconsistent elements.

The main sort aps is considered as non-observable. Thus removing the inconsistent airport schedules in $BU_V \circ BFree_S(A)$ for an arbitrary import-algebra $A \in Beh(APS-BIMP)$ by means of the Restriction functor $Restr_e$ - see [EM 90] - remains unnecessary because all APS-BEXP-algebras are behaviorally equivalent to its parameter-restriction. For more details concerning this relation see [Cor 90].

Thus the only operation left to be completely specified are SEARCH and RETURN, that "make observable" the aps-elements / terms; i.e. they show the "behaviour" of the aps-part in the observable sorts bool and dep, respectively.

Now consider the following fs / ps-module to be matched with the import of the aps-module:

$$FS / PS-BPAR = APS-BPAR$$

$$FS / PS-BEXP = APS-BIMP$$

$$FS / PS-BBOD = FS / PS-BEXP +$$

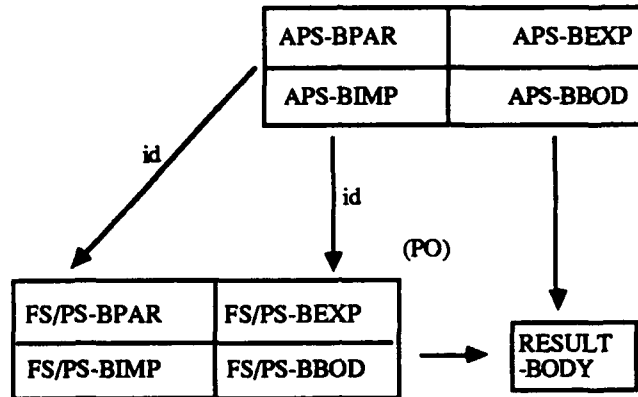
$$\text{opns} \quad \text{TAB-FS:flight\# dest dep fs} \rightarrow \text{fs}$$

$$\text{TAB-PS:plane\# type seats ps} \rightarrow \text{ps}$$

$$FS / PS-BIMP = FS / PS-BPAR$$

$$\text{eqns} \quad \dots$$

Being $FS / PS-BPAR = APS-BPAR$ and $FS / PS-BEXP = APS-BIMP$ the two modules are easy to be behaviour-composed according to definition 5.5, using the behaviour-composition-morphisms $h = (id, id)$:



The result-module is the complete behaviour aps-module, the CB-aps-module, consisting of the following components:

APS-BPAR	APS-BEXP
FS/PS-BIMP	RESULT-BODY

The above pushout satisfies the observation preserving property because the shared subpart APS-BIMP only has observable sorts; thus the composition is behaviour-consistent.

Now let us modify this example to the "view-case", (see 5.7). The same argumentation for designing the aps-scr non-observable applies to the fs- and the ps-sort, respectively, in the fs/ps-module. But if we have fs and ps non-observable in the FS/PS-BEXP- and the FS/PS-BBOD-specifications, we need to define the identity-morphism: $id: APS-BIMP \rightarrow FS/PS-BEXP$ as a view-morphism, because fs and ps are observable in APS-BIMP. The resulting module CB-aps would be a "view-module" in the sense of [Cor 90], because $v: APS-BEXP \rightarrow RESULT-BODY$ would be a view-specification-morphism.

The modules aps and fs/ps are internally correct, if the boolean part of the import-algebras is initial. Thus the usual initial constraint $INIT(BOOL)$ or view-free-generating constraint (\emptyset, i) being $i: BOOL \rightarrow SPEC$ the inclusion for the concerning specifications $SPEC$ guarantees the persistency of the behaviour-free functors from import to body. It should be placed in all parts of the modules defining a behaviour modules with constraints according to definition 6.5. Then the correctness and compositionality-result of theorem 6.7 (unit constraints) says that the CB-aps-module is again correct and its semantics is the composition of the aps-semantics and the fs/ps-semantic.

Other possibilities of formulating reasonable constraints are to find in [EM 90], 8E. Only the generating constraints we do not need anymore using the behaviour semantics.

8. REFERENCES

- [Bal 90] Baldamus, M.: Constraints and their Normal Forms in the Framework of Specification Logics (in German), Studienarbeit, TU Berlin (1990)
- [Cor 90] Cornelius, F.: Behaviour Approaches to Algebraic Module Specifications (in German), Master Thesis, TU Berlin (1990)

- [BEP 87] Blum, E.K.; Ehrig, H.; Parisi-Presicce, F.: Algebraic Specification of Modules and Their Basic Interconnections, JCSS 34,2/3 (1987), 293-339
- [Ehr 89a] Ehrig, H.: A Categorical Concept of Constraints for Algebraic Specifications; in: Categorical Methods in Computer Science - with Aspects from Topology, (H. Ehrig, H. Herrlich, H.-J. Kreowski, G. Preuß, eds.), Springer LNCS 393 (1989)
- [Ehr 89b] Ehrig, H.: Algebraic Specification of Modules and Modular Software Systems within the Framework of Specification Logics, Technical Report 89/17, TU Berlin (1989)
- [EM 85] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer (1985)
- [EM 90] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 2. Module Specifications and Constraints. EATCS Monographs on Theoretical Computer Science, Springer (1990)
- [EPO 89] Ehrig, H.; Pepper, P.; Orejas, F.: On Recent Trends in Algebraic Specification, Proc. ICALP'89, Springer LNCS 372 (1989), pp. 263-288
- [EW 85] Ehrig, H.; Weber, H.: Algebraic Specifications of Modules. Proc. IFIP Work Conf. 85: The Role of Abstract Models in Programming, Wien 1985.
- [GB 84] Goguen, J.A.; Burstall, R.M.: Introducing institutions. Proc. Logics of Programming Workshop, Carnegie-Mellon. LNCS 164, Springer (1984), 221-256
- [GR 89] Große-Rhode, M.: Parameterized Data Type and Process Specifications using Projection Algebras, in: Categorical Methods in Computer Science - with Aspects from Topology, (H. Ehrig, H. Herrlich, H.-J. Kreowski, G. Preuß, eds.), Springer LNCS 393 (1989)
- [GTW 76/78] Goguen, J.A.; Thatcher, J.W.; Wagner, E.G.: An initial algebra approach to the specification, correctness and implementation of abstract data types. IBM Research Report RC 6487, 1976. Also: Current Trends in Programming Methodology IV: Data Structuring (R. Yeh, ed.), Prentice Hall (1978), 80-144
- [Hig 64] Higgin, P.J.: Algebras with a Scheme of Operators, Mathematische Nachrichten 27 [1963/64], pp 115-132
- [HS 73] Herrlich, H.; Strecker, G.E.: Category Theory, Allyn and Bacon, Boston (1973)
- [LEFJ 91] Löwe, M.; Ehrig, H.; Fey, W.; Jacobs, D.: On the Relationship between Algebraic Module Specifications and Program Modules, Proc. TAPSOFT '91, Springer LNCS, to appear
- [Ma 89] Mahr B.: Empty Carriers: The Categorical Burden on Logic; in: Categorical Methods in Computer Science - with Aspects from Topology, (H. Ehrig, H. Herrlich, H.-J. Kreowski, G. Preuß, eds.), Springer LNCS 393 (1989)
- [NO 88] Nivela, P.; Orejas, F.: Behavioral semantics for algebraic specification languages, Proc. ADT-Workshop Gullane, 1987, Springer LNCS 332 (1988), 184-207
- [ONE 89] Orejas, F.; Nivela, P.; Ehrig, H.: Semantical Constructions for Categories of Behavioral Specifications, in: Computer Science - with Aspects from Topology, (H. Ehrig, H. Herrlich, H.-J. Kreowski, G. Preuß, eds.), Springer LNCS 393 (1989)
- [ON 90] Orejas, F.; Nivela, P.: Constraints for Behavioral Specifications, Technical Report, Univ. Politecnica de Catalunya, 1990
- [Ru 79] Rus, T.: Data Structures and Operating Systems. John Wiley & Sons (1979)
- [Ru 90] Rus, T.: Steps towards Algebraic Construction of Compilers. Technical Report, University of Iowa (1990)
- [ST 84] Sannella, D.T.; Tarlecki, A.: Building specifications in an arbitrary institution. Proc. of the Int. Symp. on Semantics of Data Types, LNCS 173, Springer (1984)
- [Tar 86] Tarlecki, A.: Software-System Development - An Abstract Viewin: H.J. Kugler (ed.): Information Proc. 86, Amsterdam, North-Holland (1986), pp. 685-688
- [TWW 78/82] Thatcher, J.W.; Wagner, E.G.; Wright, J.B.: Data type specification: parameterization and the power of specification techniques. 10th Symp. Theory of Computing (1978), 119-132. Trans. Prog. Languages and Systems 4 (1982), 711-732
- [Zi 74] Zilles, S.N.: Algebraic specification of data types. Project MAC Progress Report 11, MIT 1974, pp. 28-52

APPENDIX: GENERALIZED MORPHISMS

In this appendix we introduce the notion of generalized morphisms within the framework of a specification logic SL and discuss the connection with amalgamations in SL. This generalizes the notion of generalized homomorphisms introduced by Higgins in [Hig 64] as concept for homomorphisms between algebras of different signatures (see also [Ru 79], [GB 84], [EM 85], [Ru 90]).

In fact, generalized morphisms in SL define a category GMSL such that amalgamation is a special pushout in GMSL. Moreover, we construct general pushouts in GMSL using a generalization of Higgins Construction for the factorization of generalized morphisms. These pushouts can be considered as generalized amalgamations.

A.1 DEFINITION (Generalized Morphisms and Category GMSL)

Given a specification logic $SL = (ASPEC, Catmod)$ and objects $A_i \in Catmod(ASPEC_i)$ for $i = 1, 2$ a generalized morphism $gm: A_1 \rightarrow A_2$ is a pair $gm = (f, h)$ with

$$\begin{array}{ll} f: ASPEC_1 \rightarrow ASPEC_2 & \text{in } ASPEC, \text{ and} \\ h: A_1 \rightarrow V_f(A_2) & \text{in } Catmod(ASPEC_1) \end{array}$$

where $V_f = Catmod(f): Catmod(ASPEC_2) \rightarrow Catmod(ASPEC_1)$ is the forgetful functor (see remarks of 2.1).

The composition of generalized morphisms $gm_1 = (f_1, h_1): A_1 \rightarrow A_2$ and $gm_2 = (f_2, h_2): A_2 \rightarrow A_3$ is defined by

$$gm_2 \circ gm_1 = (f_2 \circ f_1, V_{f_1}(h_2) \circ h_1): A_1 \rightarrow A_3$$

The category **GMSL** of generalized morphisms over SL has as object class the union of all object classes of $Catmod(ASPEC)$ ranging over all objects $ASPEC$ in $ASPEC$ and the morphisms in **GMSL** are generalized morphisms.

A.2 EXAMPLES (Generalized Morphisms)

1. Given the equational specification logic EQSL then generalized morphisms are generalized homomorphisms $gm = (f, h): A_1 \rightarrow A_2$ as mentioned in [EM 85]: This means that A_i is a $SPEC_i$ -algebra for $i = 1, 2$, $f: SPEC_1 \rightarrow SPEC_2$ a specification morphism and $h: A_1 \rightarrow V_f(A_2)$ is a $SPEC_1$ -homomorphism where $V_f(A_2)$ is the $SPEC_1$ -reduct of A_2 .
2. Of course, we can also replace EQSL by the corresponding algebraic signature specification logic SIGSL. Then generalized morphisms in SIGSL are pairs of signature morphisms f and homomorphisms $h: A_1 \rightarrow V_f(A_2)$.
3. Operator schemes $\Sigma = (I, \alpha)$ in the sense of Higgins can be considered as a scheme for the construction of signatures (see [Hig 64]), where each $\omega \in \Omega$ leads to a family $(\omega_j)_{j \in J}$ of operation symbols $\omega_j: s_1 \dots s_{n_j} \rightarrow s_j$. Homomorphisms associated with a change of schemes in [Hig 64], also called general homomorphisms, are generalized morphisms over a suitable specification logic ΩSL , where abstract specifications are operator schemes Σ and $Catmod(\Sigma)$ is the category of Σ -algebras and Σ -homomorphisms in the sense of [Hig 64].
4. A specification basis $B = (I, S, \Sigma)$ in the sense of Rus (see [Ru 79], [Ru 90]) can be considered as a special case of operator schemes of Higgins (see example 3 above). Generalized homomorphisms in the sense of Rus are generalized morphisms over a suitable specification logic RSL, where an abstract specification is a specification basis $B = (I, S, \Sigma)$ and $Catmod(B)$ is the category of Σ -algebras and Σ -homomorphisms in the sense of [Ru 90].
5. For each institution INST in the sense of [GB 84] we can construct the corresponding specification logic $SL(INST)$ (see 2.2) such that generalized morphisms in $SL(INST)$ correspond to those in [GB 84].

A.3 DEFINITION (Special Morphisms and Canonical Factorization)

A generalized morphism $(f, h): A_1 \rightarrow A_2$ in **GMSL** is called standard if f is an identity in $ASPEC$ and model-identical if h is an identity such that $A_1 = V_f(A_2)$.

A factorization of a generalized morphism $(f, h): A_1 \rightarrow A_2$ into a generalized morphism (f, u) and a standard morphism (id, h^*) , i.e. $(f, h) = (id, h^*) \circ (f, u)$, is called canonical if for each other factorization $(f, h) = (id, h_2)$

Clean Algebraic Exceptions with Implicit Propagation

- extended abstract -

Pierre-Yves Schobbens*

A long-lasting stream of research strives to define a framework in which exception handling can be performed in a methodical, dependable, and concise way. Algebraic specification techniques, and specifically order-sorted algebras [Goguen and Meseguer, 1989] are our basis to meet the first two of these criteria. We tailor this basis along the lines of [Gogolla, 1987] to obtain *clean* exception algebras, to which we add implicit propagation of exceptions to reach conciseness. Our proposition meets the practical requirements of [Bidoit, 1984]:

- exception labelling: exceptions should carry information needed by the user;
- exception propagation: exceptions should propagate by default;
- exception recovery: this propagation is just a default, the user may choose to stop it and recover to a value chosen on basis of the labelling.

The strong points of our proposal are:

1. to allow a short but structured specification of exception propagation;
2. to generalize many-sorted algebras [Ehrig and Mahr, 1985] (unlike [Gogolla, 1987]);
3. to introduce a new methodological criterion, *freeness of exceptions*;
4. to give a completeness result for the associated deductive system.

1 The Problem

Algebraic specification techniques have been successfully used to specify data structures [Liskov and Zilles, 1974, Guttag, 1975], programming languages [Broy *et al.*, 1987], and are being extended to fulfill the needs of specifiers [Broy and Schobbens, 1988]. A specific problem of this approach is the treatment of exceptional situations, which has given rise to a large number of proposals (listed in the references). In many of these proposals, exceptions (sometimes called errors) are treated as supplementary values of the result sort. As a consequence, exceptions must be explicitly propagated, since there is no natural link between exceptions in the domain sorts and exceptions in the result sort. For instance, the worn-out stack example would be specified in the explicit notation of [Horebeek *et al.*, 1988] as:

```
type Stack
requires sorts elem
  operators
    underflowE: → elem
    safeE: elem → bool
  end
sorts stack
operators
  underflowS: → stack
  empty: → stack
  push: stack × elem → stack
  pop: stack → stack
  top: stack → elem
  safeS: stack → bool
variables s: stack, e:elem
```

*Unité d'Informatique, Université Catholique de Louvain, Place Sainte-Barbe, 2, B-1348 Louvain-La-Neuve (Belgique).
Tel: +32 10 47 31 50, Fax: +32 10 45 03 45, Telex: 59037 ucl b, e-mail: pys@info.ucl.ac.be
This work has been funded by the SPPS project Leibniz (RFO/AI/15)

axioms

- 1: $\text{pop}(\text{push}(s,e)) = s$ if $\text{safeS}(s)$ and $\text{safeE}(e)$
- 2: $\text{top}(\text{push}(s,e)) = e$ if $\text{safeS}(s)$ and $\text{safeE}(e)$
- 3: $\text{pop}(\text{empty}) = \text{underflowS}$
- 4: $\text{top}(\text{empty}) = \text{underflowE}$
- 5: $\text{top}(\text{underflowS}) = \text{underflowE}$
- 6: $\text{push}(\text{underflowS}, e) = \text{underflowS}$
- 7: $\text{push}(s,e) = \text{underflowS}$ if $\text{safeE}(e) = \text{false}$
- 8: $\text{pop}(\text{underflowS}) = \text{underflowS}$
- 9: $\text{safeS}(\text{underflowS}) = \text{false}$
- 10: $\text{safeS}(\text{empty}) = \text{true}$
- 11: $\text{safeS}(\text{push}(s,e)) = \text{safeS}(s)$ and $\text{safeE}(e)$

end

We see in this example that:

1. an exception (underflowE) has to be declared in the parameter, while the other (underflowS) is in the body;
2. as a consequence, "Stack" can only be used on types developed for this purpose, defeating the principles of reusability;
3. the propagation (axioms 5–8) and classification (axioms 9–11) of exceptions must be specified in both the actual parameter (not shown) and the body, resulting in a longer specification;
4. if an exception occurs in the second parameter of "push", the result is "underflowS": any information about this exception is thus lost.

Although this specification is sound, making it concise and structured requires some supplementary apparatus.

2 Definitions

We use the order-sorted algebras of [Goguen and Meseguer, 1989], which are an extension of many-sorted algebras [Ehrig and Mahr, 1985], but we do not request *local filtration*, which says (for finite signatures) that the sort hierarchy should be partitioned into components having a most general sort. This condition would amount to request separate exceptions for each data type, preventing thus exception propagation.

Signatures are extended to allow the declaration of *exceptions*; supersorts comprising declared exceptions and normal data are implicitly constructed for each normal sort, and can be designated by sort expressions.

Our exception algebras also have to satisfy an *implicit propagation* rule, that says that the leftmost unexpected exception shall be propagated.

We introduce 3 methodological constraints on the use of such algebras:

1. exception-consistency [Gogolla, 1987]: normal values and exceptions should be disjoint;
2. exception-completeness [Gogolla, 1987]: each value should be either erroneous or normal;
3. exception-freeness: exceptions should not be constrained by axioms.

A specification whose initial algebra satisfies these criteria is called *clean*.

An instance of a parameterized specification will be extended through the implicit propagation rule to deal with exceptions unknown in the context of its definition. A parameterized specification is *clean* if it is persistent on clean algebras and produces clean algebras from clean parameter algebras.

3 Example

The trivial example below shows the specification of binary trees bearing data in nodes:

```
type BinaryTree
requires sorts elem end
sorts tree
exceptions void
operators
  empty:  $\rightarrow$  tree
  node:  $\text{tree} \times \text{elem} \times \text{tree} \rightarrow \text{tree}$ 
  entry:  $\text{tree} \rightarrow \text{elem}$  raise void
```

```

    left: tree  $\rightarrow$  tree raise void
    right: tree  $\rightarrow$  tree raise void
    catch: tree raise void  $\rightarrow$  tree
variables e: elem; t1,t2: tree
axioms
    entry(empty) = void
    left(empty) = void
    right(empty) = void
    entry(node(t1,e,t2)) = e
    left(node(t1,e,t2)) = t1
    right(node(t1,e,t2)) = t2
    catch(void) = empty
    catch(t1) = t1
end BinaryTree

```

Notes:

- “tree raise void” is a sort expression designating a supersort containing tree and void. Such sort expressions can be used as domains (e.g. of “catch”) to handle exceptions.
- This specification will be augmented by new exceptions when used, in particular, exceptions of the actual parameter. These exceptions will simply be propagated.
- It is sufficiently complete for clean actual parameters only, as it treats separately normal cases (by explicit equations) and exceptions (by the implicit propagation rule).
- The side conditions present in the Stack example are now treated through the range of the variables.

4 Results

In [Schobbens, 1989], we give syntactic characterizations of *clean* specifications, and we show that equational deduction is complete for clean specifications (but not for all non locally filtered signatures).

5 Conclusions

Experience [Declerfayt *et al.*, 1988] suggests that the proposed framework allows a more systematic construction of specifications involving exceptions. Being an inessential extension of order-sorted algebras, which are themselves an inessential extension of many-sorted algebras, our proposal inherits their restrictions: among others, bounded data structures (e.g. stacks of a bounded depth) are often awkward to specify.

REFERENCES

- [Bernot *et al.*, 1986] G. Bernot, M. Bidoit, and Ch. Choppy. Algebraic semantics of exception handling. In *ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 173–186, Saarbrücken, 1986. Springer.
- [Bidoit *et al.*, 1985] M. Bidoit, B. Biebow, M.-C. Gaudel, C. Gresse, and G. Guiho. Exception handling: Formal specification and systematic program specification. *IEEE Trans. Soft. Eng. SE-11*, pages 242–251, 1985.
- [Bidoit, 1984] Michel Bidoit. Algebraic specification of exception handling and error recovery by means of declarations and equations. In *Proceedings of 11th ICALP*, volume 172 of *Lecture Notes in Computer Science*, pages 95–109, Antwerp, 1984. Springer.
- [Broy and Schobbens, 1988] M. Broy and P.Y. Schobbens. Une notation pour la spécification algébrique de systèmes concurrents. In *4e Colloque de Génie Logiciel (CGL4)*, pages 141–152. Paris, France, AFCET, 1988.
- [Broy and Wirsing, 1982] Manfred Broy and M. Wirsing. Partial abstract data types. *Acta Informatica*, 18(1), Nov 1982.
- [Broy *et al.*, 1987] Manfred Broy, Martin Wirsing, and Peter Pepper. On the Algebraic Definition of Programming Languages. *ACM TOPLAS*, 9(1):54–99, January 1987.
- [Declerfayt *et al.*, 1988] O. Declerfayt, B. Demeuse, E. Milgrom, P.Y. Schobbens, and F. Wautier. Precise standards through formal specifications: a case study: the Unix file system. In *Autumn '88 EUUG Conference*, pages 115–130, 1988.
- [Ehrig and Mahr, 1985] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification : Volume 1. Equations and initial semantics*. Springer, 1985.
- [Engels *et al.*, 1981] G. Engels, V. Pletat, and H. Ehrich. Handling errors and exceptions in the algebraic specification of data types. *Osnabrücker Schriften zur Mathematik*, 1981.
- [Gogolla *et al.*, 1984] M. Gogolla, K. Drosten, U. Lipeck, and H. Ehrich. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Computer Science*, (34):289–313, 1984.
- [Gogolla, 1987] M. Gogolla. On parametric algebraic specifications with clean error handling. In *12th Coll. Les Arbres en Algèbre et Programmation*, volume 186 of *Lecture Notes in Computer Science*, Pisa, 1987. Springer.
- [Goguen and Meseguer, 1989] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI, July 1989.
- [Goguen *et al.*, 1978] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current trends in programming methodology*, volume 4, pages 80–149. Prentice-Hall, 1978.
- [Goguen, 1977] J.A. Goguen. Abstract errors for abstract data types. In *Formal Description of Programming Concepts*, pages 21.1–21.32. MIT Press, 1977.
- [Goguen, 1978] J.A. Goguen. Exceptions and error sorts, coercion and overloading operators. Technical report, SRI, 1978.
- [Guttag, 1975] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Univ. of Toronto, 1975.
- [Horebeek *et al.*, 1988] Ivo Van Horebeek, Johan Lewi, Eddy Bevers, Luc Duponcheel, and Willy van Puymbroeck. An exception handling method for constructive algebraic specifications. *Software - Practice and Experience*, 18(5):443–458, May 1988.
- [Liskov and Zilles, 1974] B. Liskov and S. Zilles. Programming with abstract data types. In *Conference on Very High-Level Languages*, *ACM SIGPLAN Notices*, pages 50–60, 1974.
- [Poigné, 1987] Axel Poigné. Partial algebras, subsorting, and dependent types: Prerequisites of error handling in algebraic specifications. In Don Sannella and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification (5th)*, volume 332 of *Lecture Notes in Computer Science*, pages 208–234, Gullane, Scotland, Sept 1987. Springer.
- [Schobbens, 1989] P.Y. Schobbens. Declarative exception handling and propagation. Technical Report 89-18, Unité d'Informatique, Université Catholique de Louvain, 1989.

ACT TWO: An Algebraic Module Specification and Interconnection Language

Werner Fey

Technical University Berlin

Since the late sixties and early seventies abstraction concepts and structuring mechanisms are required to overcome partly the so called software crisis. Therefore, specification concepts and with them their structuring and interconnection mechanisms are developed and later integrated in some specification languages. This was done in order to write down, structure and interconnect specifications in a similar way as programming languages allow that for programs. But what is then the difference between programming languages and specifications languages or programs and specifications? The main difference makes the purpose for which programs and specifications are made: to a given problem a specification should define an abstract solution, and a program should define an efficiently running solution. This means not necessary that specifications can not be run or that programs are not abstract, but only that this is normally not the main purpose they are written for. There is a trade off between efficiently running and abstraction, but abstraction is necessary to deal with complicated, complex problems and their solutions by software. Therefore specifications are used to describe abstract solutions and to refine them step by step towards running solutions. From a theoretical point of view abstract solutions are mathematical models with algebras as special cases. So, an algebraic specification language allows to give structured formal descriptions of such models. To do that in practical cases a specification environment - very similar to a programming environment - is needed. The formal basis for its software tools should be a specification language. The implementation of such a language can be the starting point for such an environment.

There are already algebraic specification languages like CLEAR, ASL, PLUSS, or Extended ML (which have loose semantics) with modularization mechanism which closely resemble the modularization mechanism of programming languages (see /Wir 90/). ACT TWO is, unlike the specification language ACT ONE (see /EFH 83/, /EM 85/), mainly based on module specifications with its structuring and interconnection mechanisms. But the specification concept and the structuring mechanisms of ACT ONE are extended by constraints (logical formulas of first and second order) and then used to specify the constituent parts of module specifications. Therefore ACT TWO is seen as successor of ACT ONE, which is already used for the formal description of data types in the standardized specification language LOTOS for the design of distributed and concurrent systems (see /Br 88/). A specification environment for LOTOS will be built in the ESPRIT II-project LOTOSPHERE.

ACT TWO is a kernel language for the horizontal structuring of algebraic specifications of modular systems and is based on algebraic module specifications, (parameterized) construction specifications, and requirements specifications with their structuring and interconnection mechanisms. By requirements specifications some loosely defined data sets together with operations satisfying given properties are defined, perhaps by using some fixed parts defined by construction specifications. Such requirements specifications have loose semantics, i.e. all data types (algebras) satisfying the given requirements. By construction specifications fixed data sets and operations are defined perhaps over parameters given by requirements specifications. The semantics of a construction specification (with parameter) is the induced (parameterized) abstract data type, i.e. the initial / free functor semantics (see /EM 85/). In ACT TWO we use positive conditional equations for defining operations in construction specifications, first order sentences with equality for properties of operations in requirements specifications and algebraic constraints (second order logic) for generating properties of subdomains (see /EM 90/). The equations and the sentences must be given explicitly by specification expressions but the algebraic constraints are induced by the structuring and interconnection mechanisms.

Construction specifications and requirements specifications are basic specifications in ACT TWO, because both are used to formulate the components of algebraic module specifications. This is our main specification concept to define the building blocks of modular systems.

An algebraic module specification MOD consists of four components

MOD:

PAR	EXP
IMP	BOD

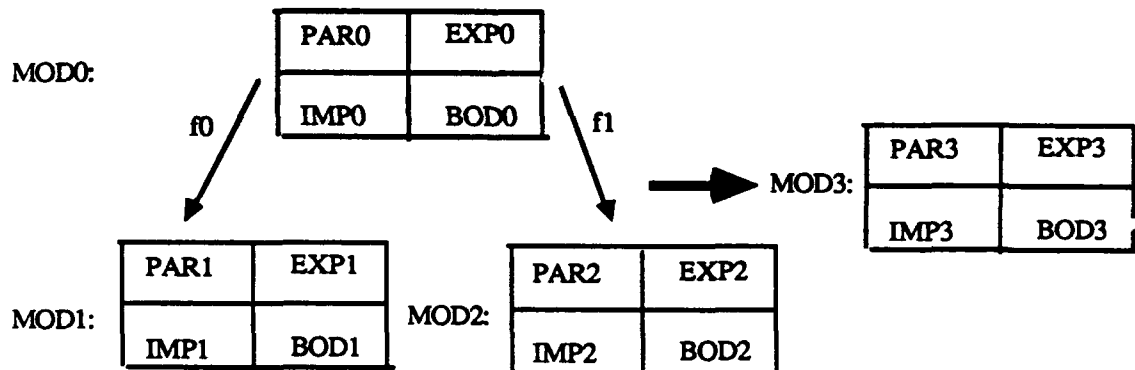
which are given by four algebraic specifications. The export EXP and the import IMP represent the interfaces of a module while the parameter PAR is a part common to both import and export and represents a part of the parameter of a modular system. These interface specifications PAR, EXP, and IMP are requirements specifications in order to be able to express requirements for operations and domains in the interfaces. The body BOD is a constructive specification, which makes use of the resources provided by the import and offers the resources provided by the export. The different parts of a module specification are connected by specification morphisms, which in most cases are inclusions.

The semantics of a module specification is given by the loose semantics of the interface specifications PAR, EXP, and IMP, a "free construction" from import to body algebras (data types), and a "behavior construction" from import to export algebras given by restriction of the free construction to the export part. A module specification is called (internally) correct if the free construction "protects" import algebras and the behavior construction transforms import algebras into export algebras (see /EM 90/).

A formal structural comparison between module specifications and program modules has been started in /LEFJ 91/.

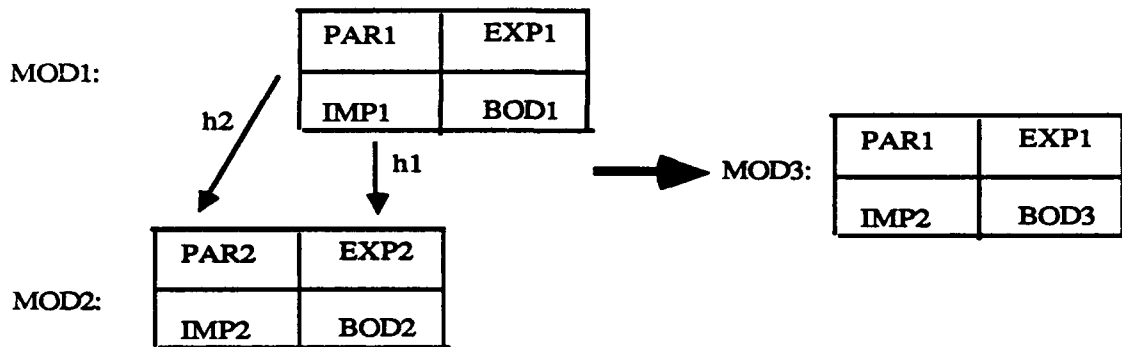
Module specifications as building blocks can be interconnected by some mechanisms for building up modular systems. The following basic interconnection mechanisms have been integrated into ACT TWO: union, composition or import actualization, (parameter) actualization, and renaming. In the following each of them gets their semantics via a module specification constructed by "flattening" the interconnection (see also /EM 90/):

The union of two module specifications MOD1 and MOD2 is a module specification MOD3 whose interfaces, body and parameter part are the union of the corresponding component specifications of the two modules. This operation allows for common parts MOD0 of the two modules to be identified:



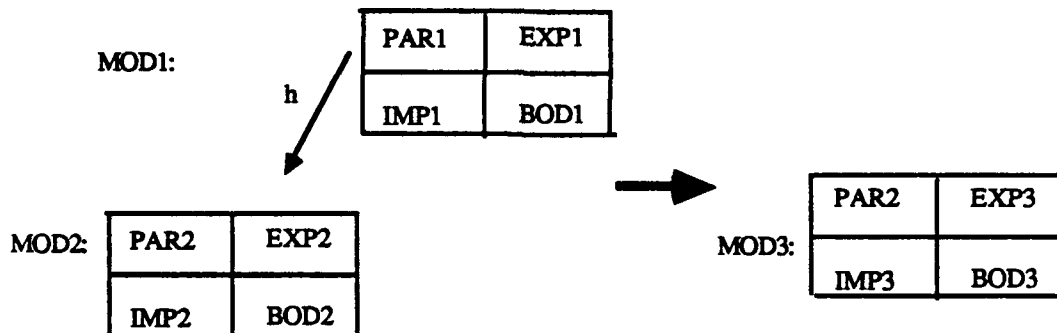
where f0 and f1 are "module specification morphisms" (i.e. 4-tuples of specification morphisms) and SPEC3 = SPEC1 + SPEC0 SPEC2 are pushout constructions for SPEC = PAR, EXP, IMP, and BOD.

The composition or import actualization of two module specifications MOD1 and MOD2 matches, via the specification morphism h_1 (and compatible h_2), the import interface of MOD1 with the export interface of MOD2. The composition yields a new module specification MOD3 whose import is the import of MOD2, whose parameter part and export interface are the corresponding ones of MOD1, and whose body is the construction specification of the body of MOD1 where, however, the import part is "exchanged" by the body of MOD2:



where $BOD3 = BOD1 +_{IMP1} BOD2$ is a pushout construction.

The (parameter) actualization of two module specifications MOD1 and MOD2 matches the parameter part of MOD1 with the export interface of MOD2 via a specification morphism h . The actualization results in a new module specification MOD3 whose export interface is that of MOD1 where the parameter part is replaced by the export of MOD2, whose parameter part is the one of MOD2, whose import part is the import of MOD2 "extended by" the import of MOD1 but without its parameter part, and whose body is the "union" of the body parts of MOD1 and MOD2:



where the module specification MOD3 has the same parameter part as MOD2, but the other parts are constructed as pushouts: $EXP3 = EXP2 +_{PAR1} EXP1$, $BOD3 = BOD2 +_{PAR1} BOD1$ and $IMP3 = IMP2 +_{IMP10} IMP12$, where however $IMP1 = PAR1 +_{IMP10} IMP12$ with $IMP10$ subspecification of $IMP2$.

Renaming of a module specification means changing names of data sorts and operations in the four component specifications but keeping the semantics; that means the renaming must be injective.

As main results for module specifications it is shown in [EM 90] that the basic interconnection mechanisms are operations on module specifications, correctness preserving, and compositional w.r.t. the semantics. This means that correctness of modular system specification can be deduced from correctness of its parts and its semantics can be composed from that of its components. Moreover, there are nice compatibility results between these operations which can be expressed by associativity, commutativity and distributivity results. This allows the restructuring of modular systems by keeping their semantics!

Similar to ACT ONE also for ACT TWO the abstract syntax is given by an extended BNF-grammar and the formal semantics is defined on two levels: The first level is that of specifications and "incomplete" specifications and the second level is that of algebras, functors and undefined objects. There is a translation from the first to the second level semantics which commutes with these two levels, i.e. the translation is an "ACT TWO-homomorphism".

For specification expressions syntactical context conditions exclude the "incomplete" specifications and the undefined objects as denotations. These context conditions - given for each rule of the BNF-grammar - are characterized by two conditions for specification texts of ACT TWO:

bottom-up consistency - all the names of specifications, operations, and data sets used in a specification text must be defined before, and

lucid library update - all specification definitions denote (syntactical complete) specifications.

If also some semantical context conditions are satisfied only (internally) correct specifications with compositional semantics are denoted. Therefore, global correctness of modular systems is implied by local correctness of the component module specifications and their interconnections. The same is true for consistency considerations (/Fey 88/).

From a theoretical point of view the basic specification concepts - requirements specification and construction specification - and their structuring mechanisms are only special cases of module specification and their interconnection mechanisms. But from a practical point of view they are necessary to describe the components of module specifications in a structured way. This is demonstrated in /Fey 88/ by specifying a part system which keeps information about products and their parts for manufacturing all kinds of technical systems.

Based on ACT TWO the type and type connection view of the specification language Π for distributed modular systems can be given a functional semantics description (see /CFGG 91/). Π is now considered as a candidate for a component description language in the EUREKA SOFTWARE FACTORY project.

References

- /CFGG 91/ J. Cramer, W. Fey, M. Goedicke, M. Große-Rhode: Towards a Formally Based Component Description Language. To appear in Proc. 4th. Int. Joint Conf. on the Theory and Practice of Software Development (Coll. on Combining Paradigms for Software Development), Brighton, April 1991
- /Br 88/ Brinksma, E. (ed.): Information processing systems - open systems interconnection - LOTOS. A formal description technique based on the temporal ordering of observational behaviour. International Standard, ISO 8807
- /EFH 83/ H. Ehrig, W. Fey, H. Hansen: ACT ONE: An Algebraic Specification Language with Two Levels of Semantics, Techn. Report No. 83-03, TU Berlin, FB 20, 1983
- /EM 85&90/ H. Ehrig, B. Mahr: Foundations of Algebraic Specifications 1 & 2: EATCS Monographs on Theor. Comp. Sci. vol. 6 & 21, Springer-Verlag 1985 & 1990
- /Fey 88/ W. Fey: Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language. Tech. Report No 88/26, TU Berlin, 1988
- /LEFJ 91/ M. Löwe, H. Ehrig, W. Fey, D. Jacobs: On the Relationship Between Algebraic Module Specifications and Program Modules. To appear in Proc. 4th. Int. Joint Conf. on the Theory and Practice of Software Development (Coll. on Combining Paradigms for Software Development), Brighton, April 1991
- /Wir 90/ M. Wirsing: Algebraic Specification. In Handbook of Theoretical Computer Science vol.B, editor J. van Leeuwen, Elsevier Science Publ. 1990, pp. 675 - 788

Towards a Theory of Binding Structures: Extended Abstract

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Binding structures enrich traditional abstract data types by providing support for representing binding mechanisms and structures with holes. They are intended to facilitate the representation and manipulation of symbolic structures such as programs, logical formulae, derivations, specifications and specification modules.

The motivation for our work on binding structures comes from an interest in building tools such as theorem provers, transformers, static analyzers, evaluators, rewriters, etc. that manipulate symbolic structures (both automatically and under interactive control). Many symbolic structures embody notions of binding. This means that many rules require hygiene side-conditions to avoid unintended modification of binding relations. Other sorts of side conditions involve occurrences, for example requiring that a variable occur at most once. It is also important to be able to describe operations or sites of rule applications in terms of sets of occurrences (paths) in a structure. Rules are often expressed informally as schemata and rule applications are often expressed in terms of allowed syntactic contexts (expressions with holes). In addition syntactic contexts are important for expressing many properties of expressions and for carrying out symbolic evaluation. Context filling is a mechanism for capturing free variables in contrast to substitution for free variables, which avoids capture. Finally, in developing efficient programs for manipulation of symbolic structures it is important to be able to express sharing and updating optimizations for algorithms and to have a clear semantics of the structures that support such refinements.

The notion of binding structure generalizes de Bruijn notation [3] and more recent theories of explicit substitution (cf. [1]) in several ways, providing for named free variables, nameless bound variables, holes, and a corresponding family of replacement operations. Binding structures capture the essence of higher-order abstract syntax, (cf. [5, 7]), but are more flexible and do not hide the structure of objects in abstract higher-order entities. Binding structures are generated from a set of binding operators. A binding operator comes equipped with a finite set of binding indices (binders), a finite set of argument selectors, and a binding relation on binder-selector pairs. Binders allow for multiple binding (keyword binding positions), argument selectors describe the arity of the operator (keyword arguments), and the binding relation specifies which binders bind in which arguments. Binding operators generalize the traditional notion of signature of an abstract data type, which has only arity information. For example lambda calculus syntax can be represented as follows. Lambda abstraction is a binding operator `lam` with binder `V`, argument selector `B`, and binding relation $\{(V, B)\}$. Application is an operator `app` with argument selectors $\{F, A\}$ and empty binders and binding relation. The `let` operator has binder `V`, argument selectors $\{B, A\}$, and binding relation $\{(V, B)\}$. Binding structures can be free variables, bound variables, holes, or binding operator applications. A free variable has an identifier and a bound variable has a binding path and binding index. The binding path determines the node in the containing structure where the variable is bound. The binding node of a bound variable may lie outside the currently known part of a binding structure. A hole has an associated substitution mapping identifiers to binding structures. Holes provide a

mechanism for trapping free variables and the associated substitution provides a means of specifying what is to be trapped and how.

The intuition is that a binding structure is a tree with edges being either selectors or identifiers, and nodes labeled by node type together with additional simple data. For example, the lambda-c expression

$$\text{lam}(\text{B} : \text{app}(\text{F} : \text{hol}(0, \{f := \text{bv}(\text{BBF}, \text{V})\}), \text{A} : \text{bv}(\text{BA}, \text{V})))$$

describes a binding tree with nodes $\{\text{mt}, \text{B}, \text{BF}, \text{BA}, \text{Bff}\}$. The root is labeled by the operator **lam** and the node **B** is labeled by the operator **app**. The node **BA** is labeled by the bound variable with binding path **BA** and binding index **V**. Following the binding path, in reverse order, up the tree we see that this variable is bound at the root by the binder **V**. The node **BF** is a hole with associated substitution mapping the identifier **f** to a bound variable with binding path **BBF**. Following this binding path up from the bound variable node in this case leads out of the tree. Hence this variable is bound externally one node above the tree under consideration. The substitution associated with the hole insures that when the hole is filled, free variables with identifier **f** will be bound, and no other free variables will be effected.¹

When a binding structure is moved relative to its external binding context, external segments of binding points must be correspondingly adjusted to preserve binding relations. $b \downarrow(p', p)$ adjusts for (rigid) motion of a structure containing b with root described by p down path p' and $b \uparrow p$ adjusts for motion up one level. For example the rearrangement $(\text{let}\{x := e_0\}e_1)e_2 \mapsto \text{let}\{x := e_0\}(e_1 e_2)$ is represented by²

$$\text{app}(\text{let}(b_0, b_1), b_2) \mapsto \text{let}(b_0 \uparrow A, \text{app}(b_1 \uparrow B \downarrow(F, \text{mt}), b_2 \downarrow(B, A)))$$

The adjustments express the fact that in this rearrangement the **let** node with b_0 at **A** and b_1 at **B** is moved up, the **app** node with b_2 at **A** is moved down along **B** and then b_1 is moved down **F**. Adjustments compose and commute according to the following equations, where b is any binding structure, p, p_0 , etc. are paths and $[p, p']$ is concatenation.

$$(\text{adj.1}) \quad (b \downarrow(p_0, p)) \downarrow(p_1, p) = b \downarrow([p_0, p_1], p)$$

$$(\text{adj.2}) \quad (b \downarrow(p_0, [p_2, p])) \downarrow(p_1, p) = (b \downarrow(p_1, p)) \downarrow(p_0, [p_2, p_1, p])$$

$$(\text{adj.3}) \quad (b \downarrow(p_0, p)) \uparrow p = b$$

There are three fundamental replacement operations on binding structures. Substitution $sb(\sigma, b)$ replaces free variables of b , using the substitution map σ from identifiers to binding structures. Unbinding $ub(\eta, b)$ replaces externally bound variables using the unbinding map η from binding points to binding structures. Filling $fil(\varphi, b)$ replaces holes of b , using the filling map φ from identifiers to binding structures. At each hole, the substitution associated with a hole is filled pointwise, and then applied to the binding structure determined by the hole's identifier and the filling map.

¹ Sequences of single character selectors and identifiers are written as strings. Each identifier in the domain of a substitution associated with a hole is considered an edge of the tree for the purpose of describing nodes, but ignored when following binding paths. Although paths are written as selector sequences to make the underlying geometry clear, what matters is the path length. Inside a binding structure paths of the same length are considered equal.

² Here and in later examples we omit selector keywords, thus $\text{let}(b_0, b_1)$ abbreviates $\text{let}(\text{A} : b_0, \text{B} : b_1)$.

Lambda binding is represented using substitution, while beta conversion is represented using unbinding. For example, using the rules for substitution, $\lambda x.f(x)$ is represented by

$$\text{lam}(sb(\{x := bv(B, V)\}, \text{app}(fv(f), fv(x)))) = \text{lam}(\text{app}(fv(f), bv(BA, V)))$$

The beta-conversion rule $(\lambda x.e)e' \mapsto e\{x := e'\}$, can be represented by

$$\text{app}(\text{lam}(b), b') \mapsto ub(\eta, b \uparrow B) \uparrow mt \quad \text{where } \eta = \{(B, V) := b'\}.$$

For example $(\lambda x.f x)y \mapsto f y$ is represented by

$$\begin{aligned} &\text{app}(\text{lam}(\text{app}(fv(f), bv(BA, V))), fv(y)) \\ &\mapsto ub(\{(B, V) := fv(y)\}, \text{app}(fv(f), bv(BA, V)) \uparrow B) \uparrow mt = \text{app}(fv(f), fv(y)) \end{aligned}$$

Using holes instead of meta-variables, we have the following simpler representation of beta-conversion.

$$\begin{aligned} l &= \text{app}(\text{lam}(\text{hol}(1, \{x := bv(B, V)\})), \text{hol}(2, mt)) \\ &\mapsto ub(\{(B, V) := \text{hol}(2, mt)\}, \text{hol}(1, \{x := bv(B, V)\})) = r \end{aligned}$$

This representation provides simple, direct expression of binding relations and hygiene conditions. Adjustments are taken care of automatically by the filling operation, they need not be accounted for explicitly in the rule. The induced conversion relation is the set B of pairs b_l, b_r such that $b_l = \text{fil}(\varphi, l)$ and $b_r = \text{fil}(\varphi, r)$ for some filling map φ . The example above is obtained using the filling map $\{2 := fv(y), 1 := \text{app}(fv(f), fv(x))\}$.

Holes can also be used to express the reduction relation generated by a conversion relation. For example b beta-reduces to b' just if there is a context c (a binding structure with a unique occurrence of a hole with identifier 0) and a conversion b_l, b_r such that $b = \text{fil}(\{0 := b_l\}, c)$ and $b' = \text{fil}(\{0 := b_r\}, c)$. Thus using the conversion above and the context $\text{lam}(\text{hol}(0, \{f := bv(B, V)\}))$ we have the following.

$$\text{lam}(\text{app}(\text{lam}(\text{app}(bv(BFBF, V), bv(BA, V))), fv(y))) \mapsto \text{lam}(\text{app}(bv(BF, V), fv(y)))$$

Let r be one of $\{sb, ub, fil\}$ and let μ_r range over r -maps (e.g. μ_{sb} ranges over substitution maps). We define composition of replacement maps uniformly as follows.

$$(\text{cmps.def}) \quad \mu_r \diamond_r \mu'_r = \mu_r \triangleleft \lambda x \in \text{Dom}(\mu'_r). r(\mu_r, \mu'_r(x))$$

where $\mu \triangleleft \mu'$ is the union of μ' and the restriction of μ that omits the domain of μ' . Then replacement map composition is associative.

$$(\text{cmps.assoc}) \quad r(\mu_r, r(\mu'_r, b)) = r(\mu_r \diamond_r \mu'_r, b)$$

Adjustment distributes across replacement. For example:

$$(\text{adj.sb}) \quad sb(\sigma, b) \downarrow (p', p) = sb(\sigma \downarrow (p', p), b \downarrow (p', p));$$

Applications of filling maps can be moved across substitutions.

$$(\text{fil.sb}) \quad \text{fil}(\sigma_0, sb(\sigma_1, b)) = sb(\text{fil}(\sigma_0, \sigma_1), \text{fil}(\sigma_0, b))$$

The adjustment and replacement operations can be expressed as a simple traversal $\text{trav}(f, d, z, b)$ where f is a function to be applied at atoms, z embodies the parameters for f and d is used to adjust z at binding operation nodes. The equations defining trav are

$$\begin{aligned} \text{trav}(f, d, z, fv(id)) &= f(z, fv(id)) \\ \text{trav}(f, d, z, bv(bpt)) &= f(z, bv(bpt)) \\ \text{trav}(f, d, z, \text{hol}(id, \sigma)) &= f(z, \text{hol}(id, \text{trav}(f, d, \sigma))) \\ \text{trav}(f, d, z, op(\dots, s : b_s, \dots)) &= op(\dots, s : \text{trav}(f, d, d(z, s), b_s), \dots) \end{aligned}$$

Thus substitution and unbinding maps lift *almost* homomorphically to general binding structures. The *almost* is due to the need for adjustment of binding points when commuting with a binding construction. This adjustment is in some sense the crucial distinction between binding structures and standard abstract data structures.

The theory of binding structures provides a natural starting point for developing generic tools for rewriting and for a more intensional treatment of syntactic structures. There are a number of features to be added to our theory in order to meet long term goals. These include: annotations; generic tools for structure walking, matching, and unification; and representation binding structures in terms of mutable structures. To add annotations one must provide mechanisms for adding annotations, for definition of alternative notions of equality (depending on how annotations are to be accounted for), and for lifting operations to annotated structures or making new annotations transparent to existing operations. A good example of the power of generic structure traversal algorithms is the "code walker" tool used in PCL, a portable implementation of the Common Lisp Object System described in [2]. This tool is used in many ways including: code analyzers, macro definitions, and transformers. The methods for representing conversion rules and reduction relations induced by a set of rules provide a simple method of embedding purely algebraic rewriting systems in languages with binding operations. The basic treatment of unification algorithms as transformations on systems of equations [4] generalizes nicely to binding structures. This provides a general framework for designing new unification algorithms and gives rise to many interesting questions. Methods of [6] are being used to develop a theory of mutable binding structures, and to develop systematic methods for refining abstract specifications into efficient implementations.

Acknowledgements. The author would like to thank Ian Mason for carefully reading early drafts and our group seminar members for many fruitful discussions. This research was partially supported by DARPA contract N00039-84-C-0211 and by NSF grants CCR-8718605 and CCR-8917606.

References

- [1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.
- [2] Pavel Curtis. Algorithms: the PCL code walker. *Lisp Pointers*, 3(1):50–61, 1990.
- [3] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
- [4] Jean H. Gallier and Wayne Snyder. Designing unification procedures using transformations: a survey. *ETACS Bulletin*, 40:273–326, 1990.
- [5] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE, 1987.
- [6] I. A. Mason and C. L. Talcott. Inferring the equivalence of (first-order) functional programs that mutate data. *Theoretical Computer Science*, to appear, 199?
- [7] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988.

Proving the correctness of algebraically specified software: Modularity and Observability issues

Gilles Bernot & Michel Bidoit

LIENS - C.N.R.S. U.R.A. 1327

Ecole Normale Supérieure

45, Rue d'Ulm - 75230 PARIS Cedex 05 France

E-mail: [bernot, bidoit] @dmi.ens.fr (Internet), or @FRULM63.BITNET (Earn)

Abstract

We investigate how far modularity and observability issues can contribute to a better understanding of software correctness. We detail the impact of modularity on the semantics of algebraic specifications and we show that, with the stratified loose semantics, software correctness can be established on a module per module basis. We discuss observability issues and we introduce an observational semantics where sort observation is refined by specifying that some operations do not allow observations. Then the stratified loose approach and our observational semantics are integrated together. As a result, we obtain a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness.

1 Introduction

A fundamental aim of formal specifications is to provide a rigorous basis to establish software correctness. Indeed, it is well-known that proving the correctness of some piece of software without a formal reference document makes no sense.¹ Algebraic specifications are widely advocated as being one of the most promising formal specification techniques. However, to be provided with some algebraic specification is not sufficient per se. A precise (and adequate) definition of what does mean the correctness of some piece of software w.r.t. its algebraic specification is mandatory. This crucial prerequisite must be first fulfilled before one can develop relevant verification methods, and try to mechanize them.

Hence the adequacy of the chosen definition of correctness has a great practical impact, and we should therefore define software correctness in conformity with actual needs. In the framework of algebraic specifications, straightforward definitions of correctness turn out to be oversimplified: most programs that must be considered as being correct (from a practical point of view) are rejected. Indeed, when the program behaves correctly, there can still exist some differences between the properties stated by the specification and those verified by the program. Here, to behave correctly means that these differences are not "observable". Consequently, more elaborated definitions of correctness, taking observability into account, should be considered.

As soon as real-sized systems are involved, both the specification and the software become large and complex. Hence the validation process becomes itself an unmanageable task. At the

¹Who would attempt to prove a theorem without providing its statement?

programming level, this problem is handled by using modular programming languages. At the specification level, algebraic specifications are split into smaller units by means of specification-building primitives. Thus, with respect to software correctness, what is needed is a framework such that the various units of the specification can be related to the various modules of the software, and such that the global correctness of the software can be established from the local correctness of each software module w.r.t. its specification module.

Thus, our claim is that modularity and observability issues are fundamental to define a practicable notion of software correctness. In this paper, we will detail various aspects related to modularity, observability, and their interactions with software correctness. We introduce a semantic framework for modular observational algebraic specifications that leads to a more adequate definition of software correctness. This is only a first step towards putting software correctness proofs in practice, but we believe that practicable proof methods can be developed on top of our approach.

We assume that the reader is familiar with algebraic specifications [20,14] and with the elementary definitions of category theory [25]. An algebraic specification SP is a tuple (S, Σ, Ax) where (S, Σ) is a signature and Ax is a finite set of Σ -formulas. We denote by $Mod(\Sigma)$ the category of all Σ -algebras, and by $Mod(SP)$ the full sub-category of all Σ -algebras for which Ax is satisfied. We will also use the following technical definition:

Definition (Minimal models):

Given a specification SP , a model $M \in Mod(SP)$ is called *minimal* if, for all $A \in Mod(SP)$, if there exists a morphism $\mu : A \longrightarrow M$, then there exists a unique morphism $\nu : M \longrightarrow A$.

Note that if $Mod(SP)$ has an initial model, then it is the unique minimal model (up to isomorphism).

2 Modularity and software correctness

In this section we shall focus on the links that can (should) be established between a modular specification and the corresponding software, implemented using a modular programming language (such as e.g. Ada, Clu or Standard ML). The problem considered is to define an algebraic semantic framework such that the various pieces of the specification can be related to the various modules of the implementation and such that the global correctness of the implementation can be established from the local correctness of each software module w.r.t. its specification module.

To better understand why and how far both the modularity of the specification and the modularity of the software interact together as well as the need for a new approach to the semantics of algebraic specifications, we shall first briefly recall the main underlying paradigm of the loose approach.

A specification is supposed to describe a future or existing system in such a way that the properties of the system (**what** the system does) are expressed, and the implementation details (**how** it is done) are omitted. Thus a specification language aims at describing **classes** of correct (w.r.t. the intended purposes) implementations (realizations). In contrast a programming language aims at describing **specific** implementations (realizations). In a loose framework, the semantics of some specification SP is a class \mathcal{M} of (non-isomorphic) algebras. Given some implementation (program) P , its correctness w.r.t. the specification SP can then be established

by relating the program P with one of the algebras of the class \mathcal{M} . Roughly speaking, the program P will be correct w.r.t. the specification SP if and only if the algebra defined by P belongs to the class \mathcal{M} .²

Let us now reexamine the above picture in a modular setting. At one hand we have a **modular** specification SP made of some specification modules $\Delta SP_1, \Delta SP_2, \dots$ related to each other by some specification-building primitives. On the other hand we have a **modular** software made of some program modules $\Delta P_1, \Delta P_2, \dots$. Assume moreover that the software structure reflects the specification structure. The problem we have to solve is the following one:

1. To define a notion of correctness such that "the program module ΔP_i is correct w.r.t. the specification module ΔSP_i " is given a precise meaning.
2. To ensure that the local correctness of each program module w.r.t. its specification module implies the global correctness of the whole software w.r.t. the whole specification.
3. To carefully study how some basic requirements about the modular development of modular software, as well as their reusability, interact with the design of the semantics of modular specifications.

It turns out that the main difficulties raised by this goal are twofold:

1. Providing a (loose) semantics to specification modules is not so easy, since from a mathematical point of view (heterogeneous) algebras do not have a modular structure.
2. If our intuition and needs about modular software development and the reuse of software modules can be easily figured out, this turns out to be a more difficult task at the level of algebraic semantics.

In the following section we shall try to provide some insight into the solution we propose and into the main ideas underlying what we call the "*stratified loose semantics*".

3 The stratified loose approach

For sake of simplicity, we shall focus on the most commonly used specification-building primitive, namely the enrichment one. Moreover, we shall assume that the modular specification SP_2 we consider is made of one specification module ΔSP that enrich only one modular specification SP_1 .

According to the loose approach, the semantics of the specification SP_1 will be defined as some class \mathcal{M}_1 of Σ_1 -algebras (where Σ_1 denotes the signature associated to SP_1). Similar notations hold for SP_2 . Since we assume that SP_2 is defined as an enrichment of SP_1 by the specification module ΔSP , we have $\Sigma_2 = \Sigma_1 \oplus \Delta \Sigma$, hence $\Sigma_1 \subseteq \Sigma_2$. Let \mathcal{U} denotes the usual forgetful functor from Σ_2 -algebras to Σ_1 -algebras.

With the help of this simple context, our intuition and needs w.r.t. the modular development of modular software can be summarized as follows [10]:

²As we will see in Section 4, this is an oversimplified picture. However, in the sequel we shall adopt this oversimplified understanding of software correctness, since it will be sufficient to study the impact of modularity. Note also that our picture does not preclude more refined views about implementations, such as the abstract implementation of one specification by another (more concrete) one [4,13], or the stepwise refinement and transformation of a specification into a piece of software [2]. This indeed is the reason why we shall speak of "realizations" instead of "implementations".

1. If some piece of software fulfills (i.e. is a correct realization of) the "large" specification SP_2 , then it must be reusable for simpler purposes, i.e. it must also provide a correct realization of the sub-specification SP_1 .
2. Any piece of software that fulfills (i.e. that is a correct realization of) the sub-specification SP_1 should be reusable as the basis of some correct realization of the larger specification SP_2 . In other words, it should be possible to implement the sub-specification SP_1 without taking care of the (future or existing) enrichments of this specification (e.g. by the specification module ΔSP).
3. It should be possible to implement the specification module ΔSP without knowing which specific realization of the sub-specification SP_1 has been (or will be) chosen. Thus, the various specification modules should be implementable **independently** of each other, may be simultaneously by separate programmer teams. Moreover, exchanging some correct realization (say P_1) of the specification SP_1 with another correct one (say P'_1) should still produce a correct realization of the whole specification SP_2 , without modification of the realization ΔP of the specification module ΔSP .

The first two requirements can be easily achieved by embedding some appropriate *hierarchical constraints* into the semantics of the enrichment specification-building primitive. Roughly speaking, it is sufficient to require the following property:

Either $\mathcal{M}_2 = \emptyset$ (in that case the specification module ΔSP will be said to be hierarchically inconsistent) or $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$.

The third requirement, however, cannot be achieved without providing a suitable (loose) semantics to **specification modules**. There is no way to take this requirement into account by only looking at the semantics of specifications. However, in an initial approach to algebraic semantics (cf. e.g. [14,12]), an initial semantics can be provided for the specification module ΔSP by considering the free synthesis functor \mathcal{F}_Δ (left adjoint to the forgetful functor \mathcal{U}). In our case, nothing ensures that this free synthesis functor \mathcal{F}_Δ exists, since we have made no assumption about the axioms of the specification. Moreover, we are looking for a loose semantics of specification modules, in order to reflect **all** correct implementation choices of these modules. The following definition provides the solution we are looking for by embedding the ideas of the initial approach into the loose one:

Definition (Stratified loose semantics):

Given a modular specification SP_2 defined as the enrichment of some modular specification SP_1 by a specification module ΔSP , the semantics of the specification module ΔSP and of the modular specification SP_2 are defined as follows:

Basic case:

If the sub-specification SP_1 is empty (hence the specification SP_2 is reduced to the specification module ΔSP), then:

- The semantics of the specification SP_2 is by definition the class of all minimal models of $Mod(SP_2)$, if any; if $Mod(SP_2)$ has no minimal model, then SP_2 is said to be *inconsistent*.
- The semantics of the specification module ΔSP is defined as being the class of all functors \mathcal{F} from the category **1** to $Mod(SP_2)$, which map the object of **1** to a minimal model of $Mod(SP_2)$.³

³As usual, the category **1** denotes the category containing only one object, which can be interpreted as a Σ_1 -algebra for an empty signature Σ_1 .

General case:

Let us denote by \mathcal{M}_1 the class of models associated to the modular specification SP_1 , according to the current definition.

- The semantics of the specification module ΔSP is defined as being the class \mathcal{F}_1^2 of all the mappings \mathcal{F} such that:

1. \mathcal{F} is a (total) functor from \mathcal{M}_1 to $Mod(SP_2)$.
2. \mathcal{F} is a right inverse of the forgetful functor \mathcal{U} , i.e.:
$$\forall M_1 \in \mathcal{M}_1 : \mathcal{U}(\mathcal{F}(M_1)) = M_1.$$

If the class \mathcal{F}_1^2 is empty, then the enrichment is said to be *hierarchically inconsistent*.

- The semantics of the whole specification SP_2 is defined as being the class of all the models image by the functors \mathcal{F} of the models of \mathcal{M}_1 :

$$\mathcal{M}_2 = \bigcup_{\mathcal{F} \in \mathcal{F}_1^2} \mathcal{F}(\mathcal{M}_1).$$

The class \mathcal{M}_2 of the models of the specification SP_2 is said to be **stratified** by the functors \mathcal{F} .

Some comments are necessary to better understand the previous definition:

- Our semantics is a true loose semantics, since it associates a class of (non-isomorphic) functors (resp. algebras) to a given specification module (resp. to a given specification). However, our semantics can also be considered as a generalization of the initial approach: if we restrict to positive conditional equations, then the free synthesis functor from $Mod(SP_1)$ to $Mod(SP_2)$ exists; under suitable additional assumptions, this functor is just one specific functor in the class \mathcal{F}_1^2 .
- It is important to note that with our loose stratified semantics, the *hierarchical constraints* mentioned above are satisfied. More precisely, as soon as the specification module ΔSP is hierarchically consistent, then we have $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$. As a consequence, both the so-called “no junk” and “no confusion” properties are guaranteed. In other words, we know that the “old” carrier sets (i.e. the carrier sets of sorts defined in SP_1) will contain no “new” value, and that “old” values who may be distinct before (in at least one model of SP_1) should not be forced to be equal by the new specification module ΔSP .
- We have chosen a pseudo initial semantics (minimal models) for the basic case (a modular specification reduced to one specification module) in order to exclude trivial models (a well-known problematic feature of loose semantics). This remark does not only apply for basic specifications, but for all modular specifications in general, since this “minimal” semantics for the basic case, combined with the hierarchical constraints induced by the stratified loose semantics for the general case, will exclude trivial carrier sets for all sorts.⁴ Moreover, such a “minimal” semantics for basic specification modules turns out to be quite adequate to specify enumerated sets (such as e.g. booleans, characters, etc.).
- So far, we have stressed that the independent implementability of each specification module is a crucial aspect of modularity. Now, we would like to stress another equally important aspect of modularity, namely the specification style point of view. Indeed, when

⁴More precisely, if we consider a sort s and if we assume that there exists some operation (or some composition of operations) which has s in its domain and a sort s' defined in a basic specification module as its codomain, then the “minimal” semantics of the basic specification module will prevent from undesired confusion of values in the carrier set of s ...

writing some specification module, a natural implicit assumption made by the specifier is that the semantics of the imported sub-specifications is preserved (i.e. this semantics is established once for all). By the way, this is exactly what is guaranteed by our stratified loose semantics: as explained above, the hierarchical constraints associated to our semantics of modularity automatically restrict the class of models of the specification SP_2 to the models that preserve the enriched sub-specifications. This contrasts with a more conventional approach, where the specifier should explicitly design the axioms of the specification unit in order to guarantee the so-called no junk and no confusion properties, i.e. to guarantee the persistency of the enrichment (and this often results in unnecessary over-specification). From our point of view, there is therefore a fundamental distinction between what we call **structured specifications**, for which the no junk and no confusion properties are **explicitly** ensured by appropriate axioms, and **modular specifications**, for which similar properties are **implicitly** ensured by an appropriate semantics. Furthermore, it is clear that the hierarchical structure of a modular specification has a deep impact on its modular semantics, while this is not the case for (persistent) structured specifications, whose semantics is not altered by flattening.

- The extension of the definition above to the case where the specification module ΔSP enriches more than one specification as well as its extension to other specification-building primitives (such as e.g. parameterization) do not raise difficult problems and is described in [9].
- It is also important to note that our definition is independent of the underlying institution [19]. Thus our stratified loose approach can be used to define the semantics of any modular algebraic specification language [33]. Moreover, our stratified loose approach can even be used in a more general framework than institutions, for instance in a framework where the *Satisfaction Condition* [19] does not hold. This is obvious since, as far as the stratified loose semantics is concerned, the existence of forgetful functors (from Σ_2 -algebras to Σ_1 -algebras, with $\Sigma_1 \subseteq \Sigma_2$) is the only requirement. Indeed, this very broad scope of the stratified loose approach will be clearly demonstrated in Section 6.

We must now point out how far our stratified loose semantics solves the problem stated in the previous section. A program module ΔP will be said to be correct w.r.t. some specification module ΔSP if and only if ΔP "defines" a functor belonging to the semantics of the specification module. From our definition, it is then obvious that the "composition" of correct software modules (i.e. the software obtained by linking together these software modules) is always a correct realization of the whole specification. Thus, the main significance of the stratified loose framework outlined in this section is that it is possible to specify and develop software in a modular way, and that the correctness of the implementation should only be established on a module per module basis. A formal theory of software reusability, built on top of our stratified loose semantics, is described in [17].

Note that the definition above is given in a very general way: we have considered all algebras, finitely generated or not. It is obviously very easy to refine our definition of the stratified loose semantics in order to consider finitely generated algebras only. Indeed, we prefer to introduce a more powerful constraint, namely the restriction to algebras finitely generated with respect to a distinguished subset of the signature called the set of *generators*.⁵ Such a constraint will guarantee that for any model, all values will be denotable as some composition of these

⁵We do not detail here the refined version of the stratified loose semantics according to this additional constraint, since the modifications to be introduced are rather obvious [9].

generators. From a theoretical point of view, an important consequence of this constraint is that *structural induction restricted to the generators* is a correct proof principle. This constraint has many practical consequences too, since reasoning by means of generators helps writing the axioms in a structured way [7]. Moreover, it can be used to avoid to overspecify some operations, as demonstrated in the following two examples:

Specifying a remove operation on sets :

To specify a remove operation on sets, the following axioms are sufficient:

$$\begin{aligned} x \in \text{remove}(x, S) &= \text{false} \\ x \neq y &\implies y \in \text{remove}(x, S) = y \in S \end{aligned}$$

Considering only finitely generated models w.r.t. the generators will ensure that, for any set S , $\text{remove}(S)$ is actually a set, reachable from the empty set by some successive insertions, and not a “junk” set.

Specifying the Euclidean division :

Similarly, to specify the division of natural numbers, the following axioms are sufficient:

$$\begin{aligned} m \neq 0 &\implies 0 \leq [n - ((n \text{ div } m) * m)] = \text{true} \\ m \neq 0 &\implies m \leq [n - ((n \text{ div } m) * m)] = \text{false} \end{aligned}$$

These axioms characterize $(n \text{ div } m)$ among all natural numbers *finitely generated w.r.t. 0 and succ* (Euclid). However, without the constraint, there are models (e.g. the initial model) where $(n \text{ div } m)$ is not reached by some $\text{succ}^i(0)$; it is only an unreachable value such that the (unreachable) remainder $[n - ((n \text{ div } m) * m)]$ returns the specified boolean values when compared with 0 and m .

In Pluss [9], the distinguished subset of generators is specified apart from the other operations of the signature and is introduced by the keyword **generated by**.

A crucial issue is obviously to know when some given specification module is *hierarchically consistent*. From a general point of view, it is well-known that this is an undecidable problem. However, we would like to point out that, in our stratified loose framework, there are two distinct grounds for hierarchical inconsistency:

- As usual, hierarchical inconsistency may result from the axioms introduced by the specification module.
- Moreover, adding “new” observations on “old” sorts will in general result in an inconsistent specification module ΔSP . This is due to the fact that, with these “new” observations, it may be possible to distinguish “old” values (i.e. to prevent them from being equal), while these “old” values could have been equal in some model M_1 of SP_1 . Hence there is no total mapping from M_1 to $\text{Mod}(SP_2)$, since this model M_1 of SP_1 cannot be extended to a model of SP_2 . A typical example of such a situation is when a “new observing operation” on an “old” sort is defined in ΔSP : for instance, if we assume that SP_1 specifies natural numbers (with $M_1 \supseteq \{\mathbb{N}, \mathbb{Z}/n\mathbb{Z}\}$), specifying a “ \leq ” operation in ΔSP will result in an hierarchically inconsistent specification module. Thus, these “observing operations” should rather have been defined in the appropriate specification module, i.e. in the specification module where the “observed” sort is defined.

Note that the latter ground for hierarchical inconsistency should not be understood as a restrictive side-effect of the stratified loose semantics, but rather as a fruitful guide in structuring large specifications into specification modules. More precisely, a specification module should be considered as a unit of specification where a sort of interest, its generators, its observers, and other appropriate operations are simultaneously defined.

As a consequence of the "hierarchical constraints" required by modularity, it is necessary to state a careful distinction between *implementable* and *not yet implementable* specification modules:

- *Implementable specification modules* will have a semantics defined accordingly to the stratified loose framework, in order to allow for a modular software development and verification process.
- *Not yet implementable specification modules* will have a more flexible semantics, in order to allow for a specification development process by stepwise refinements [8].

Such a distinction is introduced in the Pluss algebraic specification language [9,11], the semantics of which is defined following the stratified loose approach. Note that this distinction contrasts with all other specification languages developed following either the initial or the loose approach, such as ASL [37,1], OBJ2 [15] and LARCH [21], where there is only a distinction between various enrichment primitives.

4 Observability and software correctness

The paradigm used in Section 2 to introduce the stratified loose approach was obviously an oversimplified understanding of software correctness. Indeed, if software correctness (w.r.t. its formal specification) is defined in such a way, then most realizations that we would like to consider as being correct (from a practical point of view) turn out to be incorrect ones. This is illustrated by the *SET* specification given in Fig. 1.

If we consider a standard realization of *SET* by e.g. lists, we do not obtain a correct realization: this is due to the axioms expressing the commutativity of the insertion operation, which do not hold for lists. However, if we notice that indeed we are only interested in the result of some computations (e.g. membership), then it is clear that our realization of *SET* by lists "behaves" correctly. Thus, an intuitively correct realization of an algebraic specification *SP* may correspond to an algebra which is not in $Mod(SP)$. This leads to a refined understanding of software correctness: a program *P* should be considered as being correct w.r.t. its specification *SP* if and only if the algebra defined by *P* is an "observationally correct realization" of *SP*. In other words, the differences between the specification and the software should not be "observable", w.r.t. some appropriate notion of "observability".

The problem is now to specify the "observations" to be associated to some specification, and to define the semantics of such "observations" in order to obtain a framework that will capture the essence of software correctness. Up to now, various notions of observability have been introduced, involving observation techniques based on sorts [18], [36], [24], [16], [30], [26], [34], [29], [28], on operations [35], on terms [32], [23] or on formulas [31], [32]. Assuming that we have chosen some observation technique, we can specify, using this technique, that some parts of an algebraic specification are observable. An observational specification is thus obtained by adding a specification of the objects to be observed to a usual algebraic specification. The

```

spec : SET ;
    use : NAT, BOOL ;
    sort : Set ;
    generated by :
         $\emptyset : \longrightarrow \text{Set} ;$ 
        ins: Nat Set  $\longrightarrow \text{Set} ;$ 
    operations :
         $\_ \in \_ : \text{Nat Set} \longrightarrow \text{Bool} ;$ 
        del : Nat Set  $\longrightarrow \text{Set} ;$ 
    axioms :
        ins(x, ins(x, S)) = ins(x, S) ;
        ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
        del(x,  $\emptyset$ ) =  $\emptyset$  ;
        del(x, ins(x, S)) = del(x, S) ;
         $x \neq y \implies \text{del}(x, \text{ins}(y, S)) = \text{ins}(y, \text{del}(x, S)) ;$ 
         $x \in \emptyset = \text{false} ;$ 
         $x \in \text{ins}(x, S) = \text{true} ;$ 
         $x \neq y \implies x \in \text{ins}(y, S) = x \in S ;$ 
        where : S : Set ; x, y : Nat ;
end SET .

```

Figure 1: A specification of sets of natural numbers

next step is to define the semantics of these observational specifications, in such a way that our paradigm “the class of the models of some specification represents all its acceptable realizations” is correctly reflected. As explained above, some correct software could correspond to an algebra which does not satisfy all the axioms of the specification, provided that the differences between the properties of the algebra and the properties required by the specification are not observable.

There are mainly two possible ways to define the semantics of observational specifications. We can extend the class of the models of the specification SP by including some additional algebras which are “behaviourally equivalent” (w.r.t. the specified observations) to a model of $Mod(SP)$ (*extension by behavioural equivalence*, see [31], [32], [23]). In the sequel such an approach will be referred to as **behavioural semantics**. We can also directly relax the satisfaction relation, hence redefine $Mod(SP)$ (*extension by relaxing the satisfaction relation*, see [30], [29], [35]). We will call these approaches **observational semantics**.

For a comparative study of these various ways of defining the semantics of observational specifications, and of the relative expressive power of the various observation techniques mentioned above, see [5]. In the sequel we will provide a short insight into the behavioural approach, and we will point out some of its limitations. In the next section we will describe a semantics based on the observational approach.

To define a behavioural semantics we first need to define an appropriate equivalence relation \equiv_{Obs} on the class $Mod(\Sigma)$ of all Σ -algebras, also called behavioural equivalence of algebras w.r.t. the specified observations Obs [31,32]. The definition of \equiv_{Obs} depends on the observational technique in use (i.e. whether we observe sorts, operations, terms or formulas [5]). Assuming


```

spec : SET-WITH-ENUM ;
  use : LIST, NAT, BOOL ;
  sort : Set ;
  generated by :
     $\emptyset$  :  $\rightarrow$  Set ;
    ins: Nat Set  $\rightarrow$  Set ;
  operations :
     $\_ \in \_$  : Nat Set  $\rightarrow$  Bool ;
    del : Nat Set  $\rightarrow$  Set ;
    enum : Set  $\rightarrow$  List ;
  axioms :
    ins(x, ins(x, S)) = ins(x, S) ;
    ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
    del(x,  $\emptyset$ ) =  $\emptyset$  ;
    del(x, ins(x, S)) = del(x, S) ;
     $x \neq y \implies \text{del}(x, \text{ins}(y, S)) = \text{ins}(y, \text{del}(x, S))$  ;
     $x \in \emptyset = \text{false}$  ;
     $x \in \text{ins}(x, S) = \text{true}$  ;
     $x \neq y \implies x \in \text{ins}(y, S) = x \in S$  ;
    enum( $\emptyset$ ) = nil ;
     $x \in S = \text{true} \implies \text{enum}(\text{ins}(x, S)) = \text{enum}(S)$  ;
     $x \in S = \text{false} \implies \text{enum}(\text{ins}(x, S)) = \text{cons}(x, \text{enum}(S))$  ;
    where : S : Set ; x, y : Nat ;
end SET-WITH-ENUM .

```

Figure 2: A variant of the specification of sets of natural numbers

that we observe a set of formulas Φ (which is the most general case), the behavioural equivalence \equiv_Φ and the associated behavioural semantics are defined as follows:

Definition (Behavioural semantics) [32]:

Given a set of observed formulas Φ , the behavioural equivalence w.r.t. Φ , written \equiv_Φ , is an equivalence relation on $\text{Mod}(\Sigma)$ defined by:

$$A \equiv_\Phi B \text{ if and only if } \forall \varphi \in \Phi \quad A \models \varphi \Leftrightarrow B \models \varphi$$

In other words, two Σ -algebras A and B are behaviourally equivalent w.r.t. a set of observable formulas Φ , if and only if A and B satisfy the same observable formulas. The class of the behavioural models of some specification SP (with observed formulas Φ), written $\text{Beh}(SP, \Phi)$, is defined by:

$$\text{Beh}(SP, \Phi) = \{B \in \text{Mod}(\Sigma) \mid \exists A \in \text{Mod}(SP) \text{ s.t. } B \equiv_\Phi A\}$$

Now we would like to point out some limitations intrinsic to behavioural semantics. It turns out that in some cases, behavioural semantics is not powerful enough to fully capture our requirements w.r.t. software correctness: in these cases, we know of some realization that we would like to consider as being correct, but unfortunately this realization cannot be shown to be behaviourally equivalent to any of the (usual) models of the specification at hand. A typical

example of such cases arises when $Mod(SP)$ is empty (i.e. when the specification is inconsistent in the usual sense). For instance, let us consider a variant of our *SET* specification as described in Fig. 2.

What we really need for this example is to observe the following set of terms:

$$W = \{x \in S\} \cup \{t \in T_{LIST\text{-signature}}(X) \mid t \text{ is of sort } Nat \text{ or } Bool\}$$

In other words, we observe membership and some *LIST* terms but we do not observe those *LIST* terms where *enum* occurs.⁶

Obviously, the specification *SET-WITH-ENUM* is inconsistent (i.e. $Mod(SP) = \emptyset$). Consequently its class of behavioural models is empty as well, whatever the observations specified and the behavioural equivalence used. Nevertheless, a realization which represents sets by non redundant lists, *ins* being realized by *cons* (when the element to be inserted is not already in the list) and *enum* being a coercion, should clearly be considered as a correct one.

The point here is that in a behavioural approach, the existence of behavioural models depends on the existence of usual models. Indeed, behavioural semantics still rely on the usual satisfaction relation, hence behavioural consistency coincides with standard consistency. This is the reason why we shall develop in the next section an approach based on observational semantics, i.e. an approach where the satisfaction relation is redefined accordingly to the specified observations.

5 Observational specifications

In this section we develop a new framework for observational specifications, the semantics of which is based on a redefinition of the satisfaction relation. We will only consider flat or structured observational specifications, modular ones being dealt with in the next section.

As explained in the previous section, we want to reflect the following idea: some data structures are observable with respect to some observable sorts (e.g. lists are observable w.r.t. their elements via terms such as $car(L)$ or $car(cdr(L))$ etc.), but we need also to prevent from observing the results of some specific operations (e.g. if a list is obtained by enumeration of a set, $enum(S)$, it must not be observed; in particular, $car(enum(S))$, which denotes a value of an observable sort, must nevertheless be non observable). This leads to the following idea: given a specification *SP*, one defines the set of *observable sorts* $SObs$, and in addition one defines the set of "operations allowing observations" which is a subset Σ_{Obs} of the signature of *SP* (e.g. Σ_{Obs} can contain all the operations except *enum*).

5.1 Definitions

Let us first define the syntax of (flat) observational specifications.

⁶Note that for this example we observe terms and not formulas. However, as shown in [5], behavioural equivalence can be defined in a similar way as above. Moreover, whatever the definition of \equiv_{Obs} is, our counter-example remains.

Definition (Observational specifications):

- An *observation* Obs over a signature (S, Σ) is a couple (S_{Obs}, Σ_{Obs}) such that $S_{Obs} \subseteq S$ and $\Sigma_{Obs} \subseteq \Sigma$.
- An *observational signature* is a couple (Σ, Obs) such that Obs is an observation over Σ .
- An *axiom* over a signature Σ is a sentence whose atoms are equalities (between two Σ -terms of the same sort, with variables) and whose connectives belong to $\{\neg, \wedge, \vee, \Rightarrow\}$. Every variable is implicitly universally quantified.
- An *observational specification* is a couple $SP-Obs = (SP, Obs)$ such that $SP = (S, \Sigma, Ax)$ is a classical specification (i.e. Ax is a set of axioms over Σ) and Obs is an observation over the signature Σ .
- If Ax only contains equalities then $SP-Obs$ is called *equational*. If Ax only contains axioms of the form $[(u_1 = v_1) \wedge \dots \wedge (u_n = v_n) \Rightarrow (u = v)]$ then $SP-Obs$ is called *positive conditional*.

Example:

We have seen that, for the specification *SET-WITH-ENUM* given in Fig. 2, we need to observe only the terms of sort *Bool* or *Nat* where the operation *enum* does not occur. Thus, it is sufficient to declare $S_{Obs} = \{Bool, Nat\}$ and $\Sigma_{Obs} = \Sigma - \{enum\}$ in order to obtain the required set of observable terms W already mentioned in Section 4.

As usual, the notion of *observable contexts* is crucial for observability [24,30,29,23,22]:

Definition (Observable contexts):

- In general a *context* over a signature Σ is a Σ -term C with exactly one occurrence of one variable.
- Given a context C , its *arity* is $(s' \rightarrow s)$, where s' is the sort of the variable occurring in C and s is the sort of (the term) C . s is also called the (target) sort of C .
- Let (Σ, Obs) be an observational signature; the associated set of *observable contexts* is the set C_{Obs} which contains all the contexts over the signature Σ_{Obs} whose target sort belongs to S_{Obs} .
- For each observable sort $s \in S_{Obs}$, the context reduced to a variable of sort s is called "the empty context" (of sort s).

Let us now define the semantics of (flat) observational specifications.

Definition (Observational semantics):

Let $SP-Obs$ be an observational specification and Σ be its signature. Let M be a Σ -algebra and let ax be an axiom of $SP-Obs$.

- Two elements a and b of M are *observationally equal* with respect to Obs if and only if they have the same sort s and for all contexts $C \in C_{Obs}$ of arity $s \rightarrow s'$, $C(a) = C(b)$ in M (according to the usual equality of set theory). In particular observational equality on observable sorts coincides with the set-theoretic equality;⁷ for the non observable sorts, the observational equality

⁷because C_{Obs} always contains the empty contexts on observable sorts.

contains the set-theoretic equality, but there are also distinct values which are observationally equal.

- The algebra M satisfies ax with respect to Obs means that for all substitutions $\sigma : T_{\Sigma}(X) \rightarrow M$, $\sigma(ax)$ holds in M according to the observational equality (defined above) and the truth tables of the connectives.
- The algebra M satisfies $SP-Obs$ means that it satisfies all the axioms of $SP-Obs$ with respect to Obs .
- The satisfaction of observational equalities is denoted by " \models_{Obs} " and we write " $M \models_{Obs}(a = b)$ ", " $M \models_{Obs} SP-Obs$ "...

Example:

It is not difficult to show that the realization of *SET-WITH-ENUM* by non redundant lists described in the previous section satisfies the observational specification given above. For instance:

- When $x \notin S$, $enum(insert(x, S)) = cons(x, enum(S))$ is satisfied because they are equal (with respect to the set-theoretic equality) in our model; thus, they are a fortiori observationally equal.
- $insert(x, insert(y, S)) = insert(y, insert(x, S))$ is observationally satisfied (even when these two list realizations of sets are not equal with respect to the set-theoretic equality) because all contexts involving *enum* do not belong to C_{Obs} ; here, all the observable contexts C in C_{Obs} have $Set \rightarrow Bool$ as arity and the top symbol of C is necessarily " \in ".

Notation:

Given an observational specification $SP-Obs$, $Mod(SP-Obs)$ is the full sub-category of $Mod(\Sigma)$ whose objects are the Σ -algebras satisfying $SP-Obs$.

The following results are trivial:

Fact-1:

Given an observational signature (Σ, Obs) , Σ -morphisms preserve observational equalities: for all $\mu : M \rightarrow M'$, if $M \models_{Obs}(a = b)$ then $M' \models_{Obs}(\mu(a) = \mu(b))$.

Fact-2:

$Mod(SP)$ is equal to $Mod(SP-Obs)$ when $S_{Obs} = S$ (due to the empty contexts).

Fact-3:

If $SP-Obs$ is equational then the usual category $Mod(SP)$ is a full sub-category of $Mod(SP-Obs)$.

Fact-4:

More generally, if SP is an equational specification and if $Obs_1 \subseteq Obs_2$ then $Mod(SP-Obs_2)$ is a full sub-category of $Mod(SP-Obs_1)$.

Notice that, from Fact-2, Fact-3 is a particular case of Fact-4. Moreover, the inclusions stated in Fact-3 (hence in Fact-4) are often strict: there is often a model M with two elements $a \neq b$ such that $M \models_{Obs}(a = b)$.

Fact-5:

Fact-3, hence Fact-4, cannot be extended to non equational specifications. For example let M be an algebra such that $a \neq b$ and $c \neq d$ with $M \models_{Obs}(a = b)$ and $M \not\models_{Obs}(c = d)$. Then, M satisfies $[(a = b) \Rightarrow (c = d)]$ in the classical sense because the precondition is false, but it does not satisfy this axiom with respect to Obs .

When flattened, our specification of *SET-WITH-ENUM* is an example where $Mod(SP)$ only contains algebras with a trivial *Nat* carrier (a singleton), but $Mod(SP-Obs)$ contains, among others, algebras where the *Nat* part is isomorphic to \mathbb{N} .

5.2 Initiality results

As usual, initiality results can only be easily obtained for equational, or positive conditional, specifications [38].

Theorem (Least congruence):

Let $SP-Obs$ be a positive conditional observational specification and M be a Σ -algebra. There exists a least congruence \equiv on M such that the quotient algebra M/\equiv satisfies $SP-Obs$.

Sketch of the proof: Let F be the family of all the congruences such that M/\equiv satisfies $SP-Obs$. It is not empty because the trivial congruence τ defined by $(a \tau b) \Leftrightarrow (a \text{ and } b \text{ have the same sort})$ belongs to F . Let \equiv be the intersection of all the congruences in F ; if \equiv still belongs to F then the theorem is proved. Consequently, we simply have to prove that M/\equiv satisfies (observationally) each axiom of SP . This is not difficult, by applying the definitions.

The following corollary is simply obtained from the previous theorem with $M = T_\Sigma$ (as in [20]):

Corollary (Initial object):

The category $Mod(SP-Obs)$ has an initial object $I = (T_\Sigma/\equiv)$.

It may seem surprising that, regardless of several other works on observability [24,27,28], we care about initial objects while they care about terminal ones. Indeed we believe that any "collapse between values" reflects some implementation choice (each implementation choice being intuitively reflected by some equation which induces new collapses). From this viewpoint, "considering the least congruence" means "considering only the necessary implementation choices"; thus, the initial algebra can be considered as the most general realization. More generally, when the initial algebra does not exist, minimal models can be considered as realizations with "minimal implementation choices". Moreover, $Mod(SP-Obs)$ has always a terminal object which is the trivial algebra. This algebra has clearly no interest. This is due to the fact that, for the moment, our specifications are flat. Terminal algebras get interest only when some enriched specifications are "protected". We shall consider such hierarchical constraints when observational semantics and the stratified loose approach will be integrated together.

5.3 Structured observational specifications

The following proposition generalizes Fact-4 above.

Proposition:

Let $SP-Obs_1$ and $SP-Obs_2$ be two observational specifications such that $SP-Obs_1 \subseteq SP-Obs_2$. If $SP-Obs_1$ is equational then the forgetful functor \mathcal{U} from $Mod(\Sigma_2)$ to $Mod(\Sigma_1)$ has the following property:

For all Σ_2 -algebras M satisfying $SP-Obs_2$, the Σ_1 -algebra $\mathcal{U}(M)$ satisfies $SP-Obs_1$. Then \mathcal{U} also denotes the forgetful functor from $Mod(SP-Obs_2)$ to $Mod(SP-Obs_1)$.

Proof: Results from $\mathcal{C}_{Obs-1} \subseteq \mathcal{C}_{Obs-2}$.

This proposition can be extended to positive conditional observational specifications whose axioms only contains equalities of **observable sorts** (in S_{Obs}) in the **preconditions**. Such an extension is similar to some sufficient conditions used in [22,6] for proof methods with observability.

Note that, from Fact-5 above, this proposition cannot in general be extended to a non equational specification $SP-Obs_1$. Moreover, for the same reason, observational specifications do not define an institution [19] because the *Satisfaction Condition* is not guaranteed in our framework. Anyway, it seems clear that this counter-fact is intrinsic to the observability question: there is no reasonable syntactical constraint which ensures that an enrichment does not add new observations of old values. Consequently some axioms which were satisfied by the models of a specification can become unsatisfied when new observations are added. This can only be handled through modularity constraints, which cannot be easily reflected within the institution framework (just because of the *Satisfaction Condition*). As explained later on, we shall reflect these constraints owing to the stratified loose semantics.

Provided that the forgetful functor \mathcal{U} exists (from $Mod(SP-Obs_2)$ to $Mod(SP-Obs_1)$), and that $SP-Obs_2$ is positive conditional, \mathcal{U} as a left adjoint, as stated in the following theorem:

Theorem (Free synthesis functor):

Let $SP-Obs_1$ and $SP-Obs_2$ be two observational specifications such that $SP-Obs_1 \subseteq SP-Obs_2$. If $SP-Obs_1$ is equational and $SP-Obs_2$ is positive conditional then the forgetful functor \mathcal{U} admits a left adjoint functor \mathcal{F}_Δ from $Mod(SP-Obs_1)$ to $Mod(SP-Obs_2)$. In particular, there is a unit of adjunction which provides a canonical Σ_1 -morphism from M to $\mathcal{U}(\mathcal{F}_\Delta(M))$ for every algebra M in $Mod(SP-Obs_1)$.

Sketch of the proof: We use the existence of a minimal congruence exactly as for the classical ADJ framework of algebraic specifications with positive conditional axioms. For each $M \in Mod(SP-Obs_1)$ we consider the Σ_2 -algebra $T_{\Sigma_2}[M]$. Let \equiv be the least congruence on $T_{\Sigma_2}[M]$ generated by the (observational) axioms of $SP-Obs_2$ and the fibers of the canonical morphism from $T_{\Sigma_1}[M]$ to M . The $SP-Obs_2$ algebra $T_{\Sigma_2}[M]/\equiv$ is by definition $\mathcal{F}_\Delta(M)$. It is not difficult (but tedious!) to prove that \mathcal{F}_Δ is compatible with the morphisms (thus it is a functor) and that there is a natural bijection between $Hom_{SP-Obs_1}(X, \mathcal{U}(Y))$ and $Hom_{SP-Obs_2}(\mathcal{F}_\Delta(X), Y)$ for all objects $X \in Mod(SP-Obs_1)$ and $Y \in Mod(SP-Obs_2)$ (which is the definition of adjunction).

As usual, the existence of the unit of adjunction allows to define *hierarchical consistency* within an initial framework [3].

Definition (Hierarchical consistency):

Let $SP-Obs_1$ and $SP-Obs_2$ be two observational specifications such that $SP-Obs_1 \subseteq SP-Obs_2$, $SP-Obs_1$ is equational and $SP-Obs_2$ is positive conditional. The observational specification $SP-Obs_2$ is *hierarchically consistent* w.r.t. $SP-Obs_1$ if and only if the canonical morphism from I_1 to $\mathcal{U}(I_2)$ is a monomorphism (i.e. is injective in our framework), where I_1 (resp. I_2) denotes the initial algebra of $Mod(SP-Obs_1)$ (resp. $Mod(SP-Obs_2)$).

Remember that left adjoint functors preserve initial models, thus $I_2 = \mathcal{F}_\Delta(I_1)$.

Unfortunately, a similar definition of *sufficient completeness* (via the surjectivity of the canonical morphism from I_1 to $\mathcal{U}(I_2)$) is not adequate. For example, when considering our *SET-WITH-ENUM* enrichment, this canonical morphism is not an epimorphism: in the initial object I_2 , the terms $enum(S)$ create new list values which are only *observationally* equal to old list values. Thus, the following definition could be better:

Definition (Sufficient completeness):

Let $SP-Obs_1$ and $SP-Obs_2$ be two observational specifications such that $SP-Obs_1 \subseteq SP-Obs_2$, $SP-Obs_1$ is equational and $SP-Obs_2$ is positive conditional. The observational specification $SP-Obs_2$ is *sufficiently complete* w.r.t. $SP-Obs_1$ if and only if the canonical morphism μ from I_1 to $\mathcal{U}(I_2)$ has the following property:
For all values $v \in \mathcal{U}(I_2)$, there exists a value $u \in I_1$ such that $\mathcal{U}(I_2) \models_{Obs} (v = \mu(u))$.

This allows us to define *persistency*:

Definition (Persistency):

Let $SP-Obs_1$ and $SP-Obs_2$ be two observational specifications such that $SP-Obs_1 \subseteq SP-Obs_2$, $SP-Obs_1$ is equational and $SP-Obs_2$ is positive conditional. The observational specification $SP-Obs_2$ is *persistent* w.r.t. $SP-Obs_1$ if and only if it is hierarchically consistent and sufficiently complete.

Of course, such an initial approach is rather restrictive. We must more or less restrict ourselves to equational specifications in order to exploit the results stated in this section. However, almost all the classical results build on the top of the ADJ group approach are then usable. In particular, if a specification has been written following the "fair presentation" method then it is sufficiently complete, and if there are no explicit equations between generators then it is persistent [7].

Nevertheless, we believe that our definition of sufficient completeness is not fully satisfactory because I_2 does not protect the predefined data structure reflected by I_1 . Indeed, our realization of sets by non redundant lists already described is a suitable model, while I_2 is not a suitable model. This means that the initial approach is not fully adequate: hierarchical constraints should be substituted to sufficient completeness. More generally, structured specifications are probably not powerful enough to capture the essence of observability. In other words, we believe that observability issues intrinsically require a modular approach with semantic constraints.

6 Integrating observability and the stratified loose approach together: Modular observational specifications

In this section we show how we can obtain a satisfactory approach to software correctness by embedding observability into the stratified loose approach defined in Section 3.

Remember that when we have defined the stratified loose semantics of modular specifications in Section 3, we have claimed that this definition was (more than) institution independent. We will benefit here from this property, since considering modular observational specifications (with the observational semantics defined in the previous section) instead of standard modular specifications directly provides us with the adequate semantics we are looking for. To be more precise, we have explained in the previous section that the observational semantics we have introduced does not lead to an institution of observational specifications. However, since the existence of forgetful functors from Σ_2 -algebras to Σ_1 -algebras is the only requirement really needed for the stratified loose approach, there is no difficulty to translate the definition of the stratified loose semantics for modular observational specifications.

Thus, combining the stratified loose semantics (for modularity) with the observational semantics defined in Section 5 provides a framework where:

- The global correctness of some software w.r.t. its formal specification can be established on a module per module basis.
- Local correctness is defined in a way flexible enough to cope with “non observable” differences between the properties of the software module and the properties specified by the corresponding specification module.

The crucial point here is that the hierarchical constraints induced by the stratified loose semantics will guarantee us that the composition of correct software modules will always result in a correct software, and that the various modules of the specification can be implemented independently of each other. Hence we do not have to worry about the somewhat problematic features discussed at the end of Section 5. More precisely, the “no junk” and “no confusion” properties inherent to the stratified loose approach (cf. Section 3) are still valid here, and there is no need for the definitions of “sufficient completeness” and “hierarchical consistency” given in the initial approach to structured observational specifications. Moreover, in the previous section we have provided arguments for demonstrating that our observational semantics is powerful (i.e. “flexible”) enough, since the counter example discussed at the end of Section 4 was solved in an elegant way by an adequate redefinition of the satisfaction relation.

We believe that the framework developed in this paper provides a firm basis to establish the correctness of some (modular) software w.r.t. its (modular, observational) specification. However, if we really want to prove the correctness of some software, then we need adequate deduction rules and proof techniques. This point is far beyond the scope of this paper, but we would nevertheless discuss some proof related aspects. Remember that in Section 3 we have introduced the restriction to finitely generated models (w.r.t. the operations specified as generators) to guarantee that “induction w.r.t. the generators” is a correct proof principle. An obvious question is whether a similar restriction can be introduced in the framework of observational specifications, and whether we will obtain a similar powerful proof principle.

As a first remark we should note that the restriction to finitely generated models (w.r.t. the generators) is not adequate since such a restriction will be somehow contradictory with the aim of the observational semantics we have developed so far. To illustrate this we will consider the following example:

Example (Stacks implemented by arrays):

Let us consider an observational specification of stacks of natural numbers where

Σ_{Obs} is the singleton $\{Nat\}$ and Σ_{Obs} is equal to Σ ; the generators being obviously *emptystack* and *push* for the sort *Stack*. Of course, we would like to consider a model which implements stacks by means of arrays as an observationally correct model: stack values are couples (a, h) where a is an array and h is the height of the stack; *emptystack* is realized by some initial array $a = init$ and $h = 0$, *push* records its element at range h in a and increments h , *pop* simply decreases h without modifying a , etc.

Let us assume that the initial array *init* uniformly contains 0 for all indices. Then, all stacks values obtained via the generators *emptystack* and *push* satisfy the following property: for all indices $i \geq h$, $a[i] = 0$. But this property is not satisfied for the stack *pop(push(1, emptystack))* (because $h = 0$ and $a[0] = 1$ for this stack value). Consequently this model is not finitely generated w.r.t. the generators *emptystack* and *push*.

Nevertheless, one should remark that even though *pop(push(1, emptystack))* is not equal to *emptystack* according to the set-theoretic equality in our model, it is observationally equal to *emptystack*.

Thus, it is clear that we must allow values that are not denotable by a composition of generators; but we can still obtain the desired proof principle by requiring for each value to be observationally equal to a value denotable by a composition of generators. This leads to the following definition.

Definition (Observational restriction to generators):

Let *SP-Obs* be a modular observational specification. Let $\Omega \subseteq \Sigma$ be the set of generators declared in *SP-Obs*. A model M of *SP-Obs* is *observationally finitely generated w.r.t. Ω* if and only if for every value m in M there exists an Ω -term t such that $M \models_{Obs} (m = t)$.

As a second remark, we would like to point out that refining the stratified loose semantics with the "observational restriction to generators" constraint has at least two advantages. It simplifies the proof principles and moreover, it has an important consequence on the "specification style": some operations can be specified in a really abstract manner, as demonstrated in the following toy example:

Example (pickout in sets):

Let us consider a specification of sets with a *pickout* operation which is supposed to delete one of the elements of a set. We do not want to specify which element has to be deleted. This specification module is described in Fig. 3.

For sake of simplicity, let us assume that the elements are the boolean values. The models that we clearly would like to accept are the ones which contain four set values up to observational equality: \emptyset , $\{true\}$, $\{false\}$ and $\{true, false\}$. Two possible behaviours of *pickout* are acceptable: *pickout*($\{true, false\}$) = $\{true\}$ or *pickout*($\{true, false\}$) = $\{false\}$.⁸ As a matter of fact, we exactly get these models when we consider the semantic constraint of "observational restriction to generators". If this constraint is not required, then we get exotic models such as:

- The set carrier contains five values: \emptyset , $\{true\}$, $\{false\}$, $\{true, false\}$ and a strange set $\{true+false\}$.

⁸These equalities are only observational ones.

```

spec : SET-WITH-PICKOUT ;
  use : ELEM, NAT, BOOL ;
  sort : Set ;
  generated by :
     $\emptyset$  :  $\rightarrow$  Set ;
    ins: Elem Set  $\rightarrow$  Set ;
  operations :
     $_ \in _$  : Elem Set  $\rightarrow$  Bool ;
    card : Set  $\rightarrow$  Nat ;
    pickout : Set  $\rightarrow$  Set ;
  axioms :
    ins(x, ins(x, S)) = ins(x, S) ;
    ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
     $x \in \emptyset$  = false ;
     $x \in \text{ins}(x, S)$  = true ;
     $x \neq y \implies x \in \text{ins}(y, S) = x \in S$  ;
    card( $\emptyset$ ) = 0 ;
     $x \in S = \text{false} \implies \text{card}(\text{ins}(x, S)) = \text{succ}(\text{card}(S))$  ;
    pickout( $\emptyset$ ) =  $\emptyset$  ;
     $S \neq \emptyset \implies \text{card}(\text{pickout}(S)) = \text{pred}(\text{card}(S))$  ;
     $x \in \text{pickout}(S) = \text{true} \implies x \in S = \text{true}$  ;
    where : S : Set ; x, y : Elem ;
end SET-WITH-PICKOUT .

```

Figure 3: Yet another specification of sets

- *ins* is a constant function on $\{\text{true}+\text{false}\}$ which always returns $\{\text{true}+\text{false}\}$ itself and it works as usual on the other sets.
- $_ \in _$ is the constant function *true* on $\{\text{true}+\text{false}\}$ and it works as usual on the other sets.
- $\text{card}(\{\text{true}+\text{false}\}) = 1$ and the cardinal of the other sets is the usual one.
- $\text{pickout}(\{\text{true}, \text{false}\}) = \{\text{true}+\text{false}\}$ and *pickout* on all other sets returns the empty set.

This model clearly satisfies the specification. However $\{\text{true}+\text{false}\}$ is not observationally equal to a standard set because there are two distinct values (*true* and *false*) which are members of it, but its cardinal is 1.

This example, together with the *remove* and *div* examples given in Section 3, show that semantic constraints are fundamental in order to reach a specification style which is really abstract.

Obviously, the definition of correct proof principles for modular observational specifications requires further investigation. Some combination of “observational induction w.r.t. the generators” and of “context induction” à la Hennicker [22] could prove adequate.

As a last remark, we would like to remind that, as for standard modular specifications, the hierarchical inconsistency of a given observational specification module can result either from "inconsistent axioms" or from the introduction of "new" observations on "old" sorts. However, in the framework of modular observational specifications, "inconsistent axioms" and "adding new observations on old sorts" should be interpreted with respect to the observational satisfaction relation and the specified observations ΔObs . It is clear that the "flexibility" induced by observability will prove useful for hierarchical consistency issues as well. Moreover, if it is obvious that the observations should be carefully designed, this task is made easier since they are explicitly specified.

7 Specifying adequate observations

In this section we would like to hark back to our claim that the observational semantics defined in Section 5 is powerful ("flexible") enough, and to the role of those operations who prevent some observations (such as *enum*).

Our *SET-WITH-ENUM* example (cf. Fig. 2) was used to justify the need for an observational semantics. However, one could argue that this example was a bit ad hoc, since the purpose of the *enum* operation was rather mysterious: what could be the use of an operation which never provides observable results?

In general, such operations correspond to "internal services" used to define some other operations. For instance, assume that the *LIST* module provides a *sum* operation, which computes the sum of all the natural numbers contained in a list. Assume moreover that this *sum* operation belongs to Σ_{Obs} . Then we can compute the sum of all the natural numbers contained in a set by the following term: $sum(enum(S))$.

Well, things are not that easy: the term $sum(enum(S))$ is not observable. Nevertheless, this apparent difficulty can be easily solved by defining a new operation $\sigma : Set \rightarrow Nat$, in the *SET-WITH-ENUM* specification module, with the following axiom: $\sigma(S) = sum(enum(S))$. It is then sufficient to specify that σ belongs to Σ_{Obs} and we are done. Note that the resulting specification module remains hierarchically consistent, since the *sum* operation on lists is associative and commutative.

One could believe that the need for a new operation σ exhibits some weakness of our approach. On the contrary, our point is that preventing from observing the results of some operations can be considered as some kind of a very flexible visibility control mechanism. More precisely, these operations are "internal services" who can be used to define more complex computations, and as such they are available through the whole specification. However, a "client" of any realization of the specification is not allowed to directly invoke these internal services (e.g. by the term $sum(enum(S))$), but should instead invoke some explicitly made available service (e.g. σ).

8 Conclusion

We have investigated how far modularity and observability issues can contribute to a better understanding of software correctness. We have detailed the impact of modularity on the semantics of algebraic specifications. We have shown that, with the stratified loose semantics, software

correctness can be established on a module per module basis. Then we have discussed observability issues. In particular, we have explained why a behavioural semantics of observability (based on an equivalence relation between algebras) is not fully satisfactory. Therefore, we have introduced an observational semantics (based on a redefinition of the satisfaction relation) where sort observation is refined by specifying that some operations do not allow observations. Then we have integrated the stratified loose approach and our observational semantics together. As a result, we have obtained a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness. Moreover, we have shown that, with modular observational specifications, we reach a specification style which is really abstract. Our definition of software correctness is a first step towards putting software correctness proofs in practice. A promising area for further investigations is the development of (modular) proof methods on top of our approach.

Acknowledgement: We would like to thank Teodor Knapik for numerous fruitful discussions. This work is partially supported by C.N.R.S. GRECO de Programmation and E.E.C. Working Group COMPASS.

References

- [1] E. Astesiano and M. Wirsing. An introduction to ASL. In *Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations*, 1986.
- [2] F.L. Bauer et al. *The Munich project CIP. Volume I: The wide spectrum language CIP-L*. Springer-Verlag L.N.C.S. 183, 1985.
- [3] G. Bernot. Good functors...are those preserving philosophy. In *Proc. of the Summer Conference on Category Theory and Computer Science*, pages 182-195, Springer-Verlag L.N.C.S. 283, 1987.
- [4] G. Bernot, M. Bidoit, and C. Choppy. Abstract implementations and correctness proofs. In *Proc. of the 3rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 236-251, Springer-Verlag L.N.C.S. 210, 1986.
- [5] G. Bernot, M. Bidoit, and T. Knapik. *Observational approaches in algebraic specifications: A comparative study*. Technical Report 6, LIENS, 1991.
- [6] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 1991.
- [7] M. Bidoit. Algebraic data types: Structured specifications and fair presentations. In *Proc. of the AFCET Symposium on Mathematics for Computer Science*, 1982.
- [8] M. Bidoit. Development of modular specifications by stepwise refinements using the Pluss specification language. In *Proc. of the Unified Computation Laboratory*, Oxford University Press, 1991.
- [9] M. Bidoit. Pluss, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université Paris-Sud, 1989.
- [10] M. Bidoit. The stratified loose approach: A generalization of initial and loose semantics. In *Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specifications of Abstract Data Types*, pages 1-22, Springer-Verlag L.N.C.S. 332, 1987.

- [11] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? an experiment with the Pluss specification language. *Science of Computer Programming*, 12(1), 1989.
- [12] H. Ehrig, W. Fey, and H. Hansen. *ACT ONE: an algebraic specification language with two levels of semantics*. Technical Report 83-03, TU Berlin FB 20, 1983.
- [13] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209-263, 1982.
- [14] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. Equations and initial semantics*. Volume 6 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.
- [15] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, pages 52-66, 1985.
- [16] H. Ganzinger. Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems*, 5(3):318-354, 1983.
- [17] M.-C. Gaudel and T. Moineau. A theory of software reusability. In *Proc. of the European Symposium on Programming (ESOP)*, pages 115-130, Springer-Verlag L.N.C.S. 300, 1988.
- [18] V. Girratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proc. of Mathematical Foundations of Computer Science (MFCS)*, pages 576-587, Springer-Verlag L.N.C.S. 45, 1976.
- [19] J.A. Goguen and R.M. Burstall. Introducing institutions. In *Proc. of the Workshop on Logics of Programming*, pages 221-256, Springer-Verlag L.N.C.S. 164, 1984.
- [20] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An initial approach to the specification, correctness, and implementation of abstract data types*. Volume 4 of *Current Trends in Programming Methodology*, Prentice Hall, 1978.
- [21] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in five easy pieces*. Technical Report 5, Digital Systems Research Center, 1985.
- [22] R. Hennicker. *Context induction: A proof principle for behavioural abstractions and algebraic implementations*. Technical Report MIP-9001, Fakultät für Mathematik und Informatik, Universität Passau, 1990.
- [23] R. Hennicker. Implementation of parameterized observational specifications. In *Proc. of TAPSOFT*, pages 290-305, Springer-Verlag L.N.C.S. 351, 1989.
- [24] S. Kamin. Final data types and their specification. *ACM Transactions on Programming Languages and Systems*, 5(1):97-123, 1983.
- [25] S. Mac Lane. *Categories for the working mathematician*. Volume 5 of *Graduate Texts in Mathematics*, Springer-Verlag, 1971.
- [26] J. Meseguer and J.A. Goguen. *Initiality, induction and computability*, pages 459-540. *Algebraic Methods in Semantics*, Cambridge University Press, 1985.

- [27] L.S. Moss, J. Meseguer, and J.A. Goguen. Final algebras, cosemicomputable algebras and degrees of unsolvability. In *Proc. of Category Theory and Computer Science*, pages 158–181, Springer-Verlag L.N.C.S. 283, 1987.
- [28] L.S. Moss and S.R. Thate. Generalization of final algebra semantics by relativization. In *Proc. of the 5th Mathematical Foundations of Programming Semantics International Conference*, pages 284–300, Springer-Verlag L.N.C.S. 442, 1989.
- [29] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specification. In *Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specification of Abstract Data Types*, pages 184–207, Springer-Verlag L.N.C.S. 332, 1987.
- [30] H. Reichel. Behavioural validity of conditional equations in abstract data types. In *Contributions to General Algebra 3, Proc. of the Vienna Conference*, 1984.
- [31] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. In *Proc. of TAPSOFT*, pages 308–322, Springer-Verlag L.N.C.S. 185, 1985.
- [32] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specification revisited. *Acta Informatica*, (25):233–281, 1988.
- [33] D.T. Sannella and A. Tarlecki. Building specifications in an arbitrary institution. In *Proc. of the International Symposium on Semantics of Data Types*, Springer-Verlag L.N.C.S. 173, 1984.
- [34] Oliver Schoett. *Data abstraction and the correctness of modular programming*. PhD thesis, University of Edinburg, 1987.
- [35] N.W.P. van Dieppen. Implementation of modular algebraic specifications. In *Proc. of the European Symposium on Programming (ESOP)*, pages 64–78, Springer-Verlag L.N.C.S. 300, 1988.
- [36] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.
- [37] M. Wirsing. *Structured Algebraic specifications: A kernel language*. PhD thesis, Techn. Univ. Munchen, 1983.
- [38] M. Wirsing and M. Broy. Abstract data types as lattices of finitely generated models. In *Proc. of the 9th Symposium on Mathematical Foundations of Computer Science (MFCS)*, 1980.

A Formal Approach to Software Testing

Extended Abstract April 15, 1991

Gilles Bernot²
bernot@frulm63
bernot@dmi.ens.fr

Marie Claude Gaudel¹
mcg@frlri61
mcg@lri.lri.fr

Bruno Marre¹
marre@frlri61
marre@lri.lri.fr

Introduction

Software testing is sometimes considered as intrinsically empirical, without possibility of theoretical grounds. There is some evidence that it is not the case: see for instance [25] and [26] for some accounts of the research in the area and [14] or [15] for some theoretical foundations.

This paper addresses the problem of the formalization of software testing. More precisely, we are concerned with the use of formal specifications for guiding the testing activity. It is well-known that one of the advantages of formal specifications is to give the possibility of correctness proofs of programs. We believe that formal specifications also provide a basis for what we call “formal testing”: i.e. the definition of testing in a formal framework which makes explicit the relationship between testing and correctness, as well as some complementarity of testing and proving.

We present briefly here some new developments of a research activity which has been reported in [6, 7, 8, 11, 21, 22], and more recently in [1] and [3]. Our previous results were specialized to positive conditional algebraic specifications. It turns out that it is possible to work in the more general framework of institutions [12]. It makes it possible to consider a larger class of formal specifications. Moreover, it makes clearer the relation between the semantics of the formal specifications and the corresponding testing process.

1 Formal specifications and correctness

Briefly, a formal specification method (institution) is given by a *syntax* and a *semantics*.

- The syntax is defined by a notion of *signature*; with each signature Σ is associated a set of *sentences* Φ_Σ . Φ_Σ contains all the well-formed formulas built on Σ , some variables, some logical connectives and quantifiers.
- The semantics is defined by a class of Σ -interpretations, Int_Σ , and a *satisfaction relation* on $Int_\Sigma \times \Phi_\Sigma$ denoted by \models . For each Σ -interpretation A and for each formula ϕ , “ $A \models \phi$ ” should be read as “ A satisfies ϕ ”.

In this framework, a *formal specification* is a pair $SP = (\Sigma, Ax)$ such that Ax is a (finite) subset of Φ_Σ .

The class of interpretations *satisfying* SP is called the class of *models* of SP and is denoted by $Mod(SP)$:

$$Mod(SP) = \{ A \in Int_\Sigma \mid A \models Ax \}$$

¹LRI, UA CNRS 410, Université PARIS-SUD, F-91405 Orsay cedex.

²LIENS, URA CNRS 1327, Ecole Normale Supérieure, 45 rue d’Ulm, F-75230 Paris cedex 05.

The notions of signature, sentence, interpretation, and validation depend on the kind of formal specification, for instance equational algebraic specifications, temporal logic,...

Let SP be a formal specification and P be a program. It is possible to verify (by testing or by proving) the adequacy or inadequacy of P with respect to SP if the semantics of P and SP are expressible in some common framework. As we are interested in dynamic testing, this role is ensured here by the concept of an interpretation for a given signature: intuitively a Σ -interpretation is a set of values plus, for each name in the signature Σ , an operation of the relevant arity on these values. We consider that P defines a Σ -interpretation M_P . Then, the question of the correctness of P with respect to SP becomes: does M_P belong to the class of models of SP ?

2 Testing a program against a property

In our framework, a *test data* is a Σ -formula $\phi(X)$. As said above, $\phi(X)$ is a well-formed composition of logical connectives, operation names of Σ , and variables in X . A test data ϕ is *executable* by a program P if ϕ is a ground formula and if P actually defines a Σ -interpretation, i.e. P provides an implementation of every operation of the signature. Under these conditions, *running a test* ϕ consists of computing by M_P the operations of Σ which occur in ϕ and checking that the results returned by M_P satisfy the property required by the connectives.

This view of program testing is just a generalization of the classical way of running tests, where the program is executed for a given input, and the result is accepted or rejected: in this case, the formula is the input-output relation required for the program.

There are some cases where it is not possible to decide whether or not an execution returns a correct result, mainly for reasons of insufficient observability of M_P : a property required by the specification may not be directly observable using the program under test (see for instance [16]). It is an aspect of the so-called *oracle problem*: the oracle is some decision process which should be able to decide, for each test data ϕ whether ϕ is successfully run or not when submitted to the program P . Providing such an oracle is not always trivial ([23, 24]) and may be impossible for some test data.

3 Exhaustive data sets, Hypotheses

Let us come back to the notion of correction and the satisfaction relation. As an example of an institution, let us consider equational algebraic specifications as defined in [13].

Example 1: In [13] the satisfaction relation is stated as

“ A Σ -equation is a pair $e = \langle L, R \rangle$ where $L, R \in T_\Sigma(X)$, for some sort s . Let $var(e) = var(L) \cup var(R) \dots$ A Σ -algebra A satisfies e iff $\bar{\theta}(L) = \bar{\theta}(R)$ for all assignments $\theta : Y \rightarrow A$ where $Y = var(e)$. If E is a set of Σ -equations, then A satisfies E iff A satisfies every $e \in E$ ”.

This definition naturally leads, for a given specification, to the following test data set:

Definition: given an equational algebraic specification $SP = \langle \Sigma, E \rangle$, the *exhaustive* test data set, $Exhaust_{SP}$, is the set of all ground instances of all the equations in E .

$$Exhaust_{SP} = \{\sigma(\phi) \mid \phi \in E, range(\sigma) = T_\Sigma\}$$

where \mathcal{T}_Σ is the set of ground terms on Σ .

This test set is exhaustive with respect to the specification, not with respect to the program, since we limit the assignments to those values of M_P which are denotable by some term of \mathcal{T}_Σ . It means that the success of this test set will ensure correctness only if M_P is finitely generated with respect to Σ . Thus we have introduced an additional *hypothesis* on M_P (the first one was that M_P is a Σ -interpretation). We note this hypothesis $Adequate_\Sigma(M_P)$ ³. Let us assume that we have a correct oracle procedure named *success*; in this institution, as “=” is the only predicate, the oracle is a procedure which is able to decide, given two values of M_P , whether or not they represent the same abstraction in SP (cf. the *identify* step in [10]). Now, we have, assuming $Adequate_\Sigma(M_P)$:

$$success(Exhaust_{SP}) \iff M_P \in Mod(SP)$$

$Exhaust_{SP}$ is obviously not usable in practice since it is generally infinite. One way to make it practicable, i.e. finite, is to make stronger hypotheses on the behaviour of M_P . These *testing hypotheses* represent and formalize common test practices; for instance, identifying subdomains of the variables where the program is likely to have the same behaviour; in this case, it is no more necessary, assuming the hypothesis, to have all the ground instances of the variables, but only one by subdomain. As an example, it is commonly assumed, when testing a stack implementation, that its behaviour, when pushing a value on a given stack, then popping, does not depend on the value. Such hypotheses are called *uniformity hypotheses*. There are other possible patterns of hypotheses, for instance *regularity hypotheses*, etc [3].

The choice of these hypotheses is driven by the specification. For instance, it is shown in [22] and [9] how unfolding conditional axioms can be used to identify uniformity subdomains. Besides, this choice also depends on the acceptable size of the test set: as usual in testing, the problem is to find a sound trade-off between cost and quality considerations.

4 Validity and Unbias

The hypotheses we have mentioned have the property that, starting from the axioms of the specification, they allow to select executable test sets which are *valid* and *unbiased*.

Assuming some hypotheses H and the correction of an oracle procedure *success*, a test set T is *valid* if:

$$success(T) \implies M_P \in Mod(SP)$$

T is *unbiased* if, assuming H :

$$M_P \in Mod(SP) \implies success(T)$$

Validity expresses that assuming the hypothesis, uncorrect programs are detected; unbiased is the converse property, i.e. correct programs are not rejected.

A sufficient condition to ensure validity and unbiased is to select test sets which are *subsets* of the exhaustive test set, *derived from the hypotheses*. However, this limitation is sometimes too strong. For instance, the need of an oracle may lead to build an observable test data

³It is not surprising to have such assumptions in a specification-based testing approach: they just express that the program under test, which is seen as a “black box” should not be too far from the specification.

set which is not a subset of the exhaustive data set (it is however, valid and unbiased); or it is sometimes efficient to perform some simplifications of the formulas of the data set (see [3] sections 2.3 and 2.4.)

Summing up, the theoretical framework presented here introduces the important idea that a test data set cannot be considered (or evaluated, or accepted, etc) independently of some hypotheses and of an oracle. In [3] and [1], we define a *testing context* as a triple (H, T, O) where T is the test data set, H is a set of hypotheses and O an oracle. We then construct a basic testing context (*Adequate_Σ, the axioms, the undefined oracle*) and we provide some way for deriving from it, by successive refinements, *practicable* testing contexts. A testing context (H, T, O) is practicable if: T is finite; O is defined on all the test data in T ; and, assuming the hypotheses H , it rejects no correct programs (*unbias*) and accepts only correct programs (*validity*). Unbias and validity are ensured by the fact that the refinement always results in a subset of one of the exhaustive test sets mentioned above and in Section 5.

5 Changing of Institution

We have shown on Example 1 how to derive, in the “ADJ institution” a triple (*minimal hypothesis, exhaustive test set, equality oracle*) in a canonical way. In the sequel we will call this triple $(Hmin_{ADJ}, Exhaust_{ADJ, SP}, Eq_{ADJ})$. It is interesting to look at such derivations for some other semantic approaches of algebraic specifications.

Example 2 (operational semantics of algebraic specifications):

Let us consider that the semantics of an algebraic specification $SP = \langle \Sigma, E \rangle$ is the term rewriting system obtained from E by orientation of the equations⁴. The corresponding definition of satisfaction is:

$$M_P \models E \iff (\forall t \in \mathcal{T}_\Sigma, M_P \models t = t\downarrow)$$

$t\downarrow$ being a normal form, i.e. we consider the usual reflexive, symmetric and transitive closure of the one-step rewriting relation, on ground terms.

This leads to the following exhaustive test set for a specification SP :

$$Exhaust_{TRS, SP} = \{t = t\downarrow \mid t \in \mathcal{T}_\Sigma\}$$

and the minimal hypothesis $Hmin_{TRS}$ is that M_P is a Σ -interpretation. It is possible to use the same patterns of testing hypotheses, i.e. uniformity and regularity, as for Example 1.

The basic testing context is defined by $Hmin_{TRS}$, $Exhaust_{TRS, SP}$, and the oracle procedure can be provided by an ASSPEGIQUE-like [4] or an OBJ-like system [19] in conjunction with an equality decision, similarly to the oracle of Example 1.

Example 3 (observational semantics of algebraic specifications):

We consider here that the class of models of $SP = \langle \Sigma, E, Obs \rangle$ is the class $Beh(SP)$ as defined by [18, 16, 2] and others. Obs is a subset of the sorts of SP , called the observable sorts.

The notion of *observable contexts* is crucial for observational semantics: an

⁴When this TRS is confluent and terminating, this ensures the existence of at least one model.

observable context over a sort s is a Σ -term C of observable sort, with exactly one occurrence of one variable of sort s . Then, a Σ -algebra A satisfies an equation $e = \langle L, R \rangle$ iff $C[\bar{\theta}(L)] = C[\bar{\theta}(R)]$ for all assignments θ and all observable contexts C over the sort of e .

Then the observational exhaustive test set for a specification SP is $Exhaust_{OBS,SP}$:

$$\{C[t] = C[t'] \mid t, t' \in s, (t = t') \in Exhaust_{ADJ,SP}, C \text{ observable context over } s\}$$

and $Hmin_{OBS} = Adequate_{\Sigma}$.

This institution is interesting for our testing purpose since it provides a way of solving the oracle problem with weak hypotheses: let us take as Obs the set of predefined sorts (with equality) of the programming language. It is sensible to assume that these equality implementations are correct (however, it remains a hypothesis.) Then the oracle decision just use these equalities. This approach is reported in [1] and [3].

More generally, in order to apply our formal testing approach, the institution under consideration must provide a way of defining the exhaustive test set $Exhaust_{SP}$ as well as a way of characterizing a correct oracle *success*. Roughly speaking, the institution should provide us with a notion of "ground formulas" canonically derived from a set of axioms (the exhaustive test set), as well as a practicable definition of the satisfaction of ground atoms and some satisfaction rules associated with the connectives. Notice that this can be difficult, for instance if existential quantifiers are allowed in the specifications.

6 Testing and proving

One of the possible continuation of this work may be the definition of a verification framework where proving and testing would complement each other, in a way similar to proofs and refutations in mathematics [20]. A natural idea is to prove the hypotheses. It is not clear that it is always worth the effort: in Example 3, proving the oracle hypothesis can be very costly, and not truly necessary; on the contrary, $Adequate_{\Sigma}$ can be proved easily if the programming language provides a well-defined encapsulation mechanism; some uniformity hypotheses could be verified by static analysis of the program. This point deserves further reflections: this implies to have a common theoretical background for testing and proving. Note that our ideas in this area are very tentative; they are reported in the following part of this paper for the purpose of opening a discussion.

A first obvious difference between proving and testing is that a proof works on the text of the program (let us call it P) and that a test exercises the system obtained from P , i.e. in our framework M_P . However, it is possible to exhibit some similarities. Let us consider a proof method i.e. some set of inference rules allowing to deduce facts of the form:

$$\vdash correctness(P, SP)$$

where $correctness(P, SP)$ is the so-called "correctness theorem", or the "proof obligation" associated with the development of P from SP [17], we can consider that it is an oracle. The set of properties required by SP (or a subset of it) can be directly considered as a part of the test set occurring in our testing contexts. In this case, we assume some "Birkoff-like" hypothesis for each axiom ϕ of SP , namely:

$$\vdash correctness(P, \phi) \iff M_P \models \phi$$

This hypothesis means that it is assumed that both the correctness theorem and the execution support of P (i.e. translators, operating system) are consistent with the semantics M_P of P ... Similar issues are studied in the perspective of software proving in the ProCoS research project [5].

Conclusion

We propose a formalization of black-box testing, i.e. of those testing strategies which do not depend of the program structure, but on its specification only. This approach is based on formal specifications and it seems to be rather general and applicable to a significant class of formalisms. Moreover, we have introduced a notion of testing hypothesis which expresses the gap between the success of a test and correctness, and which could bring a way of combining testing and proving.

As soon as specifications are handled in a formal framework and the testing strategies are expressed as hypotheses (i.e. formulas), one can use proof-oriented tools for the selection of test data sets. These tools depend on the kind of formal specification in use: in [22] and [3] we present a tool based on "Horn clause logic" which allows to deal with algebraic specifications and produces test sets corresponding to the combination of some general hypothesis schemes. Our tool is even more elaborated since it helps in the choice of hypotheses.

Acknowledgements

This work has been partially supported by the Meteor Esprit Project and the PRC "Programmation et Outils pour l'Intelligence Artificielle".

References

- [1] Bernot G. *Testing against formal specifications : a theoretical view*. TAPSOFT CCPSD 91, LNCS 467, Brighton, April 1991.
- [2] Bernot G., Bidoit M. *Proving the correctness of algebraically specified software: Modularity and Observability issues*. In this conference AMAST-2, TCS, may 22-25, Iowa city, Iowa, USA, 1991.
- [3] Bernot G., Gaudel M.-C., Marre B. *Software Testing based on Formal Specifications: a theory and a tool*. Software Engineering Journal, to appear, 1991.
- [4] Bidoit M., Choppy C., Voisin F. *The ASSPEGIQUE specification environment, Motivations and design*. Proc. of the 3rd Workshop on Theory and Applications of Abstract data types, Bremen, Nov 1984, Recent Trends in Data Type Specification (H.-J. Kreowski ed.), Informatik-Fachberichte 116, Springer Verlag, Berlin-Heidelberg, 1985, pp.54-72.
- [5] Bjorner D. *A ProCos Project Description*. EATCS Bulletin, October 1989
- [6] Bougé L. *A proposition for a theory of testing: an abstract approach to the testing process*. Theoretical Computer Science 37, 1985.
- [7] Bougé L. Choquet N. Fribourg L. Gaudel M. C. *Test sets generation from algebraic specifications using logic programming*. Journal of Systems and Software Vol 6, n°4, pp.343-360, November 1986.

- [8] Choquet N. *Test data generation using a PROLOG with constraints*. Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, pp 132-141, July 1986.
- [9] Dauchy P., Marre B. *Test data selection from the algebraic specification of a module of an automatic subway*. LRI report n° 638, LRI, Université Paris-sud, Orsay, France, January 1991, to appear in proc. ESEC 91.
- [10] Ehrig H., Kreowski H.-J., Mahr B., Padawitz P. *Algebraic implementation of abstract data types*. Theoretical Computer Science, Vol.20, pp. 209-263, 1982.
- [11] Gaudel M.-C., Marre B. *Algebraic specifications and software testing: theory and application*. LRI Report 407, Orsay, February 1988, and extended abstract in Proc. workshop on Software Testing, Banff, IEEE-ACM, July 1988.
- [12] Goguen J.A., Burstall R.M. *Introducing institutions*. Proc. of the Workshop on Logics of Programming, Springer-Verlag L.N.C.S. 164, pp.221-256, 1984.
- [13] Goguen J. Thatcher J. Wagner E. *An initial algebra approach to the specification, correctness, and implementation of abstract data types*. Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978.
- [14] Goodenough J.B., Gerhart S.L. *Towards a theory of test data selection*. IEEE trans. soft. Eng. SE-1, 2, 1975. Also: SIGPLAN Notices 10 (6), 1975.
- [15] Gourlay J.S. *A mathematical framework for the investigation of testing*. IEEE Transactions on Software Engineering, vol. SE-9, n° 6, November 1983.
- [16] Hennicker R. *Observational implementation of algebraic specifications*. Acta Informatica, vol.28, n°3, pp.187-230, 1991.
- [17] Jones C. B. *Systematic software development using VDM*. Second edition, Computer Science Series, Prentice Hall, C.A.R. Hoare ed., 1990.
- [18] Kamin S. *Final Data Types and Their Specification*. ACM Transactions on Programming Languages and Systems (5), pp.97-123, 1983.
- [19] Kirchner C., Kirchner H., Meseguer J. *Operational semantics of OBJ3*. Proc. of the 15th International Colloquium on Automata, Languages and Programming (ICALP), Springer-Verlag L.N.C.S. 317, pp.287-301, 1988.
- [20] Lakatos I. *Proofs and Refutations, The logic of mathematical discovery*. J.Worral and E.Zahar ed., Cambridge University Press, 1976.
- [21] Marre B. *Génération automatique de jeux de tests, une solution : Spécifications algébriques et Programmation logique*. Proc.Programmation en Logique, Tregastel, CNET-Lannion, pp.213-236, May 1989.
- [22] Marre B. *Toward automatic test data set selection using Algebraic Specifications and Logic Programming*. Proc. Eighth International Conference ICLP'91 on Logic Programming, Paris, June 25-28, 1991.
- [23] Weyuker E. J. *The oracle assumption of program testing*. Proc. 13th Hawaii Intl. Conf. Syst. Sciences 1, pp.44-49, 1980.
- [24] Weyuker E. J. *On testing non testable programs*. The Computer Journal 25, 4, pp.465-470, 1982.

[25] Proc. 1st ACM-IEEE Workshop on Software Testing, Banff, July 1986.

[26] Proc. 2nd ACM-IEEE Workshop on Software Testing, Verification, and Analysis, Banff, July 1988.

A Case Study Towards Algebraic Verification of Code Generation (Extended Abstract)

Heinrich Hussmann

Institut für Informatik
Technische Universität München
Postfach 202420
W-8000 München 2
Germany

E-Mail: hussmann@lan.informatik.tu-muenchen.de

This paper reports on a small but typical case study which covers the use of algebraic methods for the verification of code generation for applicative languages.

In our approach, both the source and the target language are represented algebraically. The abstract syntax is seen as an abstract datatype containing an evaluation function, specified by equations. The compiler itself is given by equations, too, defining a function from source syntax to target syntax. Altogether, three operations are specified algebraically:

eval_src:	$\text{Source_Syntax} \times \text{Input} \rightarrow \text{Output}$
eval_tgt:	$\text{Target_Syntax} \times \text{Input} \rightarrow \text{Output}$
compile:	$\text{Source_Syntax} \rightarrow \text{Target_Syntax}$

The verification task now is, pretty naively, equivalent to a proof of the theorem:

$$\forall i \in \text{Input}, s \in \text{Source_Syntax}: \\ \text{eval_tgt}(\text{compile}(s), i) = \text{eval_src}(s, i)$$

(Please note that this view, in contrast to the method described in a series of papers from (Morris 73) to (Rus 90), does not require the source and target languages to have a similar algebraic structure; neither the compiler has to work homomorphically.)

The case study presented here does not aim at the theoretical background for code verification. Instead, it describes experiences when using an existing software tool set for algebraic specifications on the problem.

The three components (source language, target language, compiler) are specified for a small but typical language (see below), using the input language of the system RAP (Geser, Hussmann 86). In the full version of this paper, it is demonstrated that this system can help to validate a compiler by providing a tool for systematic generation of test cases. It is explained how the verification task itself can be performed interactively with the TIP system (Fraus, Hussmann 90), a theorem prover using an extended version of structural induction. It is shown how a user can derive from the system's output some hints for inventing appropriate auxiliary lemmata.

As the source language for these experiments, elements of a simple applicative language have been chosen, containing arithmetic operations for integer numbers, case analysis, input and output statements, and recursive function declarations.

The target language is a simple stack machine. A complete specification of such a pair of languages and an appropriate compiler has been given in (Hussmann, Rank 89). In (Schmidt 90) the verification with the help of TIP is described, covering all language constructs except of recursive function calls.

For the purposes of this presentation, characteristic problems are discussed only for three constructs of such a source language:

- arithmetic expressions (as an introductory example)
- case analysis (as a classical and typical example)
- recursive function calls (as the missing construct in (Schmidt 90)).

The following piece of specification (for arithmetic expressions as an example source code) gives an impression of the used style. Specifications are built in a modular way on top of other specifications (indicated by the keyword **basedon**). The keyword **cons** introduces a special kind of operations which are interpreted as constructors for the resp. target sorts.

```
type EXP
basedon OP, NAT
sort Exp
cons const: (Nat) Exp,
      binop: (Op, Exp, Exp) Exp
func eval_src: (Exp) Nat
axioms all (op: Op, n: Nat, e1, e2: Exp)
  eval_src(const(n)) -> n,
  eval_src(binop(op,e1,e2)) ->
    evalOp(op,eval_src(e1),eval_src(e2))
endofstype
```

An arithmetic expression here either is a natural number (expressed by the constructor **const** for "constant") or it is made of a binary operation (**binop**) applied to two other expressions. The function **eval** is a classical recursive evaluator for such an expression. The operators are listed in a separate specification called **OP**. A simple choice for **OP** is:

```
type OP
basedon NAT
sort Op
cons plus, times: Op
func evalOp: (Op,Nat,Nat) Nat
axioms all (n1, n2: Nat)
  evalOp(plus,n1,n2) -> add(n1,n2),
  evalOp(times,n1,n2) -> mult(n1,n2)
endofstype
```

In a similar manner, the target language and its semantics (**eval_tgt**) are defined. The function **eval_tgt** has an additional parameter for the local stack of the target machine.

The compiler essentially transforms an expression into postfix form. The compiler can be described again by a few equations in a specification which makes use of both the source and the target language:

```
type COMPILER
basedon OP, EXP, NAT, INSTR, SEQINSTR
func compile: (Exp) SeqInstr
axioms all (e, e1, e2: Exp, op: Op, n: Nat)
  compile(const(n)) -> < LOAD(n) >,
  compile(binop(op,e1,e2)) ->
    compile(e1) & compile(e2) & BINOP(op)
endofstype
```

Here the specification INSTR is assumed to contain the machine instructions of the target machine, SEQINSTR describes sequences of such instructions, where $\langle_ \rangle$ is the embedding of an element into a sequence and $_ \& _$ is the concatenation of sequences.

Using the system TIP, an attempt can be made to prove interactively the essential theorem

$$\forall (e \in \text{Exp}, i \in \text{SeqInstr}, s \in \text{Stack}) : \\ \text{eval_tgt}(\text{compile}(e), s) = \text{eval_src}(e)$$

This proof is successful only if an important lemma (which is a generalization of the theorem) is given to the system:

$$\forall (e \in \text{Exp}, i \in \text{SeqInstr}, s \in \text{Stack}) : \\ \text{eval_tgt}(\text{compile}(e) \& i, s) \\ = \text{eval_tgt}(i, \text{push}(\text{eval_src}(e), s))$$

Given this information, TIP is able to complete the proof (by structural induction) automatically.

Even if the construct of case analysis is added to the source language (and jumps to the target language), TIP suffices to achieve a complete, well-structured proof within a reasonable time (i.e. after the addition of a few, rather obvious, lemmata).

The situation becomes much more complicated, if recursive function definitions are added to the language.

A first observation is that the semantics of both languages are no longer well-defined, if the extension is made in a naive way. It is not sufficient to simulate the usual recursive evaluation rules which are known from denotational semantics. It can be easily shown (even with the help of TIP) that the presence of nonterminating recursion leads to non-standard elements for the primitive data type NAT. Therefore, the class of models where the standard induction principle on natural numbers can be applied becomes empty!

As a consequence, the semantics for algebraic specifications must be changed to incorporate partial operations. Unfortunately, this change also induces a slight change of the term rewriting calculus built into TIP, in order to keep soundness of the derivations (simulating a kind of call-by-value strategy).

Even after these changes, it turns out that structural induction is not sufficient to prove the central correctness theorem. The main reason for this problem is that the algebraic term denoting a recursive call cannot contain the code for the function body as a literal subpart.

Despite of these more negative observations, the verification problem can be slightly reformulated in such a way that the tool TIP performs all technical computations which are necessary for the correctness proof. The full version of this paper shows two ways for such a reformulation:

Approach 1: The central theorem to be proven is rewritten into a conditional clause, simulating a kind of fixpoint argument. Basically, this means to add a precondition to the correctness theorem which assumes the correctness for the body expressions of recursive definitions. Unfortunately, TIP is not yet able to handle conditional theorems in general; but the proof of this particular theorem can be completed by a simple trick (adding the precondition as a lemma which is "assumed to be correct" by the system).

Approach 2: A kind of computational induction is admitted using auxiliary counters for the number of nested recursive calls, which force all function calls to terminate after a finite number of expansions. This leads to a change of the specifications for the source and target language into new specifications which do no longer require the framework of partial operations. The proof then can be lead along a lexicographic combination of the recursion counter and the expression to be evaluated.

The mathematical basis for both proof ideas cannot be justified within the proof tool, as it is today. However, this way the technical (and most error-prone) work of the verification can be left over to the system.

As a summary, the available tool set RAP/TIP, which was designed for general algebraic specifications without the compiler aspect in mind, seems to provide a serious help for the task of verifying non-trivial compilers. On the other hand, a number of particular proof techniques, which are necessary for the treatment of recursion, are not yet fully supported. Moreover, the current tools are typical prototypes, mainly with respect to their user interface. So the development of a tool assisting in the practical development of reliable compilers still is an important research challenge – the general aim, however, seems to be realistic.

References:

(Fraus, Hussmann 90)

U. Fraus, H. Hussmann, Term induction proofs by a generalization of narrowing. To appear in: Proc. of the IMA Conf. on The Unified Computation Laboratory, Stirling, 1990

(Geser, Hussmann 86)

A. Geser, H. Hussmann, Experiences with the RAP system - a specification interpreter combining term rewriting and resolution. In: Proc. ESOP 86 Conference, Springer (LNCS 213), 1986

(Hussmann, Rank 89)

H. Hussmann, C. Rank, Specification and prototyping of a compiler for a small applicative language. In: M. Wirsing, J. A. Bergstra (Eds.), Algebraic methods: Theory, tools and applications. Springer (LNCS 394), 1989

(Morris 73)

F. L. Morris, Advice on structuring compilers and proving them correct. In: Proc. ACM Symposium on Principles of Programming Languages, Boston, 1973

(Rus 90)

T. Rus, Algebraic alternative for compiler construction. To appear in: Proc. of the IMA Conf. on The Unified Computation Laboratory, Stirling, 1990

(Schmidt 90)

H. Schmidt, On the semi-automatic verification of a compiler assisted by the TIP system. Diploma thesis (in German), Universität Passau, 1990

A Two Level Scanning Algorithm

John Knaack and Teodor Rus
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242

1 Introduction

Most of the features of a programming language that must be recognized by a scanning algorithm can be described by regular expressions, and most automatic scanner generators use some form of regular expressions for their specification. However, most programming languages also include some features that cannot be described by regular expressions, and the convenience of specifying these features is often the factor that determines the ease of use of a scanner generator. The most powerful scanner generators available, such as Rex [Gros89] and Lex [Lesk75], rely on the user writing C or Modula-2 code in order to handle these special features. It is our view that these features should be included in a scanner generator without the user needing to fall back on writing code for the specification. The Two Level Scanner (*TLS*), as discussed in this paper, accomplishes this goal.

In addition, the theory of context/noncontext computation developed for the algebraic compiler [Rus88, LePe90] provides a useful completion of the concept of context used in other scanners. Instead of specifying right context and a set of start states for left context, as some earlier scanners do [Gros89, Lesk75], the *TLS* allows the specification of both left and right context or left and right noncontext.

This scanner is called a two level scanner because it is built from a First Level Scanner, (*FLS*), which can be automatically generated from a given set of regular expressions. The particularity of these regular expressions is that they do not depend upon the programming language that is being implemented. The *TLS* works exclusively with the tokens recognized by the *FLS*, thus allowing us to extend the notion of a regular language as a lexicon specifier to that of a regular language with conditions on the lexical entities. The specification of a scanner in this framework consists of a set of easy-to-construct equations in the extended language of conditions, specifying each construct needed in the language being implemented.

2 Two Level Scanning

The lexical entities which are universally used by all programming languages are identifiers (I), integer numbers (N), floating point numbers (F), white spaces (W), unprintable characters (U), and other characters (O). All these lexical entities can be specified by regular expressions. Therefore, we decided to build the *TLS* on top of the alphabet $\{I, N, F, W, U, O\}$. Thus, any construct of a given programming language can be seen as a two level abstraction:

at the first level it is a stream of characters of the alphabet of the programming language, and at the second level it is a stream of tokens in the alphabet $\{I, N, F, W, U, O\}$. The *FLS* takes as input a stream of characters in the alphabet of the programming language and groups them into the above six classes common to all programming languages. At each call *FLS* returns a tuple of attributes $\langle Token, Lex, Len \rangle$ called a lexical entity, where the attribute $Token \in \{I, N, F, W, U, O\}$, *Lex* is the string of characters mapped into *Token*, and *Len* is the length of *Lex*. These lexical entities are kept in a buffer, and any backtracking is done on this buffer. The *TLS* calls the *FLS* for the next lexical entity and evaluates equations defining the lexicon of the programming language processed. The goals achieved with this approach are maximizing the efficiency of the scanner by scanning the source program only once [Heur86] and replacing tables of disjoint sets of characters by a six letter alphabet, thus simplifying the specification.

3 Lexicon Specification by Conditional Equations

Each class of lexical constructs of the programming language under consideration is specified by a conditional equation of the form *String* = *Lexical Specification*. The *String* is the token name of the class and *Lexical Specification* is constructed from *conditions* and *conditional expressions* which are defined by the following rules:

Conditions:

1. A condition is a relation on the attributes of the current lexical entity; for example $Token = I$, $Len \leq 8$, or $Lex = "do"$.
2. A condition is a logical expression on conditions constructed with the logical operators *or*, *and* and *not*; for example, $Token = I$ and $(Len > 3$ or $Lex = "aa")$.

The first relation occurring in a condition must specify the *Token* attribute. All following relations in that condition may then specify the *Lex* or *Len* attributes, which will refer to the same lexical entity as the *Token* attribute. Thus, the condition $Token = I$ and $(Len > 3$ or $Lex = "aa")$ in the example in the definition above states that the next lexical entity must be an identifier ($Token = I$), and the condition will be true only if the length of the identifier is more than 3 or the identifier is "aa".

Conditional expressions: A conditional expression is formed from conditions by the operations of concatenation, choice ($|$), or Kleene star ($*$), such as:

$(Token = O$ and $Lex = "." | $Token = O$ and $Lex = ","$ $Token = O$ and $Lex = ","$) $*$$

to denote zero or more occurrences of either a period or two commas.

Lexical specification: Now, a *Lexical Specification* is defined in terms of the keywords **begin**, **body**, **end**, **recursive**, **context**, **noncontext** as follows, where square brackets are used to indicate optional components:

Lexical Specification =

```
[begin Conditional Expression,]
body Conditional Expression,
[end Conditional Expression [recursive],]
[context {(Conditional Expression, Conditional Expression)},]
[noncontext {(Conditional Expression, Conditional Expression)}] ;
```

The *TLS* uses as input lexical entities delivered by the *FLS* and recognizes second level constructs of the language following their specification rules. The left-hand side of a conditional equation is the name of a class of constructs to be recognized by the scanner. The *Lexical Specification* of that conditional equation expresses the constructs of that class in terms of the conditions over the alphabet generated by the *FLS*. The *TLS* then returns a triple $\langle \textit{Name}, \textit{Lexeme}, \textit{Length} \rangle$, where *Name* is the name of the class of constructs recognized, *Lexeme* will be the concatenated strings from the *Lex* attribute of the lexical entities processed, and *Length* is the length of the attribute *Lexeme*. For example, a C comment may be specified by the following equation:

```
"comment" = begin : Token = O and Lex = "/" Token = O and Lex = "*",
               body : any*,
               end : Token = O and Lex = "*" Token = O and Lex = "/";
```

In this case, if the source contained the characters */*A comment*/*, the *TLS* would return the triple $\langle \textit{comment}, \textit{/*A comment*/, 13} \rangle$. In addition, if this type of construct were recursive, as comments are in Modula-2, the keyword **recursive** would be added after the **end** specification.

If a construct requires context or noncontext in order to be properly recognized, the keywords **context** and **noncontext** are also provided. Only one or the other can be specified. Following this keyword is a list of pairs of constant regular expressions defining the left and right context or noncontext. This syntax follows exactly the form of context and noncontext computed by the Language Analysis System [RuMa90] which prepares a grammar for the algebraic compiler. The first element of each pair specifies the construct's left context, while the second element specifies the construct's right context. The special keyword **none** can appear as the only left or right context, indicating no checking of context, while the keyword **bof** (beginning of file) can be included as the first constant of the left context, and **eof** (end of file) can be the last constant of the right context, indicating beginning of file and end of file, respectively, as context. For example, a Pascal program identifier and the final period of a Pascal program can be specified as:

```
"progid" = body : Token = I,
            context : (bof Token = I and Lex = "procedure", none);
"period" = body : Token = O and Lex = ".",
            context : (none, eof) ;
```

4 Implementation of a TLS

A Two Level Scanner is generated by an LR parser called *map*. *Map* takes as input a context-free grammar that describes the lexical specifications, along with semantic rules for constructing the *TLS*, as shown in Figure 1.

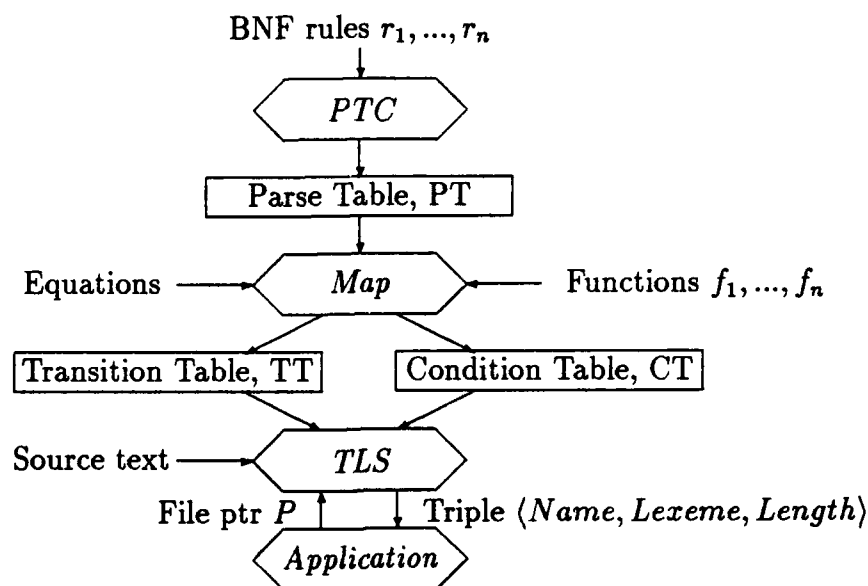


Figure 1: Constructing the *TLS*

The Parse Table Constructor (*PTC*) is an LR parser generator that builds *map* from the BNF rules r_1, \dots, r_n for the lexical specifications. The mapping of the regular expressions to the machine uses the functions f_1, \dots, f_n associated with the rules r_1, \dots, r_n . The conditional expressions are mapped into an *NFA* with a boolean stack machine attached to each state for evaluating the logical expressions used in the conditional equations. Thus, the transition performed by the next-state function of the *NFA* is conditional. This constructs a *TLS* consisting of the *NFA* resulting from the conditional equations, a Transition Table (*TT*) specifying the behavior of the *NFA*, and a Condition Table (*CT*) containing the conditions to be satisfied in order for transitions to take place. In other words, if state i has a condition $c \in CT$ on its stack machine and the state j is the next state of i in the *TT*, then the *TLS* machine will only move to state j if the condition c evaluates to True while processing the current lexical entity.

This scanner is used by the *init* program in the the *TICS* [Rus90] algebraic compiler. The *init* program generates the initial data bases for this compiler. During this step of the algebraic compiler a source program is partially translated by discovering all of its lexical entities and mapping them into the appropriate table of these data bases.

The *TLS* can be easily interfaced to any system, as the only parameter it requires is the current file pointer of the source program file, and all it returns is the tuple showing the next construct discovered by it.

References

- [Gros89] Grosch, J., Generators for High-Speed Front-Ends, LNCS, **321**:80-92.
- [Heur86] Heuring, V.P., The Automatic Generation of Fast Lexical Analysers, *Software - Practice and Experience* **16**:9, 801-808, (1986).
- [LePe90] LePeau, J., Bounded Context Parsing, Ms Thesis, The University of Iowa, Department of Computer Science, December 1990, Iowa City, IA 52242.
- [Lesk75] Lesk, M.E., LEX - A Lexical Analyser Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
- [Rus88] Rus, T., Parsing Languages by Pattern Matching, *IEEE Transactions on Software Engineering*, **14**:4, 498-510, 1988.
- [Rus90] Rus, T., Steps Towards Algebraic Construction of Compilers, Technical Report LITP90.69, Université Paris 6-7, France.
- [RuMa90] Rus, T., C.R.Maxson, Language Analysis System, Unpublished paper, The University of Iowa, Department of Computer Science, Iowa City, IA 52242.

Deriving Incremental Implementations from Algebraic Specifications

E.A. van der Meulen

CWI, P.O.Box 4079, 1009 AB Amsterdam, The Netherlands (email: emma@cwi.nl)

Abstract

We present a technique for deriving incremental implementations from algebraic specifications, belonging to the subclass of conditional well-presented primitive recursive schemes. We use concepts of the translation of well-presented primitive recursive schemes to strongly non-circular attribute grammars, storing results of function applications and their parameters as attributes in an abstract syntax tree of the first argument of the function in question. An attribute dependency graph is used to guide incremental evaluation. The evaluation technique is based on a leftmost innermost rewrite strategy. The technique is extended to *conditional* well-presented primitive recursive schemes. The class of well-presented primitive recursive schemes is a very natural one for specifying static semantics. Allowing conditions to equations in a primitive recursive scheme is the first step in extending this class to one in which the dynamic semantics of languages can be described as well.¹

Introduction

Courcelle and Franchi-Zannettacci proved that any well-presented primitive recursive scheme with parameters is equivalent to a strongly non-circular attribute grammar [CFZ82a, CFZ82b]. The result of a so called *incremental* function is interpreted as a synthesized attribute of the sort of its first argument, and the other arguments, *parameters*, are interpreted as the inherited attributes of the same sort. Primitive recursive schemes, in turn, are a subset of algebraic specifications. Following this route we can transfer techniques developed for attribute grammars to algebraic specifications. In particular we can transfer techniques developed for incremental evaluation of attributes in a dependency graph to incremental evaluation of terms in an algebraic specification.

Our method is to be implemented as part of the term rewriting engine of the ASF+SDF system, a programming environment generator. From a language definition written in the algebraic specification formalism ASF+SDF [BHK89], a program environment is generated, at the moment consisting of a syntax directed editor and a term rewriting system for evaluating terms in the editor. We want to be able to mix incremental evaluation and normal evaluation. So, the incremental evaluation strategy must be an adaptation of the usual leftmost innermost rewrite strategy. Therefore, we do not want to use the full translation of algebraic specifications to attribute grammars but only the attribute concept of Courcelle and Franchi-Zannettacci and deduce attribute dependencies directly from a specification. Thus we are able to mix incremental evaluation of terms with normal evaluation.

Note: This is an extended abstract, the complete paper is available as [Meu90].

¹Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments, phase 2 - GIPEII) and from the Netherlands Organization for Scientific Research - NWO, project *Incremental Program Generators*.

Well-presented Primitive Recursive Schemes

We illustrate our incremental implementation of well-presented primitive recursive schemes with an example. Example 1 presents part of an algebraic specification of the typechecker of a simple programming language. This specification is a well-presented primitive recursive scheme. The typecheck functions, *tcp*, *tcs* and *tcd* are the *incremental* functions. A program consists of declarations and statements, typechecking a program yields a Boolean value, when declarations are typechecked a *type-environment* is constructed: i.e. a table with identifiers and their types. This type-environment is used for typechecking the statements. Hence *TYPE-ENV* is a *parameter* of the function *tcs*. In the equations is described how a program is typechecked and how statements are typechecked.

Example 1

functions:

program	: DECLS # STATS	-> PROGRAM
decls	: DECL # DECLS	-> DECLS
stats	: STAT # STATS	-> STATS
tcp	: PROGRAM	-> BOOL
tcd	: DECLS	-> TYPE-ENV
tcs	: STATS # TYPE-ENV	-> BOOL

equations:

$$[1] \quad tcp(program(DeclS, Stats)) = tcs(Stats, tcd(DeclS))$$

$$[2] \quad tcs(stats(Stat, Stats), Type - env) = and(tcs(Stat, Type - env), tcs(Stats, Type - env))$$

When the term $tcp(program(decls(pair(x, natural)), stats(assign(x, 5)), stats(assign(x, 13))))$ is reduced, the first subterm, the program tree, is stored, and decorated with attributes: the synthesized attributes *tcp*, *tcd* and *tcs* are added to the top node, the declaration node, and the statements node, respectively. At the statement node an inherited attribute, *t-env*, is added for the parameter of *tcs*: *Attribute dependencies* can be derived from the equations. For instance, from equation [1] we deduce that attribute *tcp* depends directly on attribute *tcs*, and the parameter attribute of *tcs* depends directly on the attribute *tcd*. From equation [2] we deduce that attribute *tcs* at the *stats*-node depends directly on the *tcs* attributes of the two child nodes, and that the parameter attributes, *t-env*, at these child nodes depend directly on the parameter attribute of the parent node. Patching all these dependencies brings forth the dependency graph as shown in Figure 1.

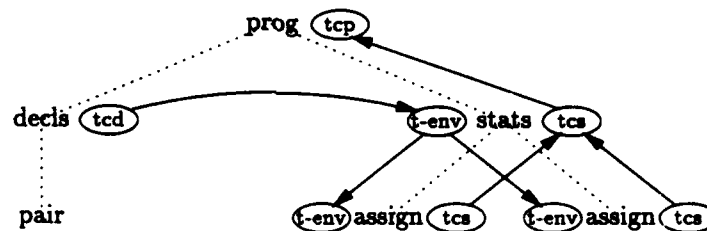


Figure 1: Top of an attributed tree

While reducing the term the normal forms of the terms $tcp(program(...))$, $tcs(stats(...), ...)$ and $tcd(decls(...))$ are stored in the corresponding synthesized attributes. Also, the inherited attributes obtain a value.

When subsequently, we want to reduce a term that is slightly different from the previous one, say, $tcp(program(decls(pair(x, natural)), stats(assign(x, 88)), stats(assign(x, 13))))$, we replace the tree $assign(x, 5)$ in the stored tree by $assign(x, 88)$ and start updating the attribute values following the

dependency graph. If a synthesized attribute has to be re-evaluated, a term is constructed out of the corresponding function, the subtree the attribute is attached to, and the values of its parameter attributes. While reducing this term, we use the normal forms stored in other attributes. Reps, Teitelbaum and Demers have described an optimal time algorithm for updating attribute values in the context of attribute grammars [RTD83]. We adapted their algorithm to an algorithm for updating attributes in the context of algebraic specifications.

Conditional Well-presented Primitive Recursive Schemes

In a non-conditional primitive recursive scheme each equation has a unique left-hand side. In a conditional primitive recursive scheme several equations may have the same left-hand side. Whereas in the non-conditional case an attribute dependency graph of a term can be constructed *before* evaluation, when using a conditional primitive recursive scheme the attribute dependency graph of a term is constructed *upon* evaluation. In this way, only the attribute dependencies of equations that are actually applied in the reduction, (i.e. whose conditions succeed) are added to the graph.

Conditions may concern attributes as well as the abstract syntax tree. If a condition concerns a subtree of the abstract syntax tree a special synthesized attribute, called *inc*-attribute, is added to the top node of this subtree. An *inc*-attribute never has a value but if the subtree is modified, successor attributes of the *inc*-attribute are re-evaluated.

Example 2 shows the equations commonly used to describe the evaluation of *if-statements*. The function *eval* is an incremental function, with parameter *VALUE-ENV*: a table of identifiers and their values.

Example 2

$$\begin{array}{l}
 [3] \quad \frac{\text{Exp} = \text{true}}{\text{eval}(\text{if}(\text{Exp}, \text{Stats1}, \text{Stats2}), \text{Value} - \text{env}) = \text{eval}(\text{Stats1}, \text{Value} - \text{env})} \\
 [4] \quad \frac{\text{Exp} = \text{false}}{\text{eval}(\text{if}(\text{Exp}, \text{Stats1}, \text{Stats2}), \text{Value} - \text{env}) = \text{eval}(\text{Stats2}, \text{Value} - \text{env})}
 \end{array}$$

When reducing the term $\text{eval}(\text{if}(\text{true}, \text{stats}(\text{assign}(\text{x}, 5)), \text{stats}(\text{assign}(\text{y}, 99))), \text{env}(\text{pair}(\text{x}, 0), \text{pair}(\text{y}, 7)))$ the first attributed tree of Figure 2 is constructed with the dependencies of of equation [3]. When reducing the term $\text{eval}(\text{if}(\text{false}, \text{stats}(\text{assign}(\text{x}, 5)), \text{stats}(\text{assign}(\text{y}, 99))), \text{env}(\text{pair}(\text{x}, 0), \text{pair}(\text{y}, 7)))$, the subtree *true* is replaced by the subtree *false*. The *inc*-attribute induces a re-evaluation of the *eval*-attribute at the *if*-node. First the incoming edges of this attribute are removed from the dependency graph. Equation [4] is applied to reduce the new *eval*-term, so, the dependencies of this equation are added, resulting in the second attributed tree of Figure 2.

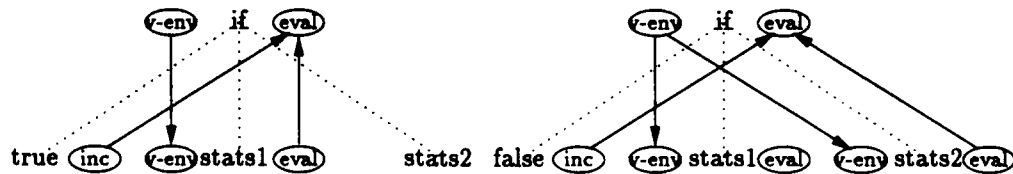


Figure 2: Attributed trees for the evaluation of an if-statement

Conclusions

Many algebraic specifications of static semantics of languages meet the requirements of a well-presented primitive recursive scheme. Conditional well-presented primitive recursive schemes provide more flex-

ibility, however, and are a first step towards obtaining incremental implementations for a more comprehensive class of algebraic specifications. Existing ASF+SDF-specifications of the typechecking of ASPLE, mini-ML, and Pascal fall within the class of conditional well-presented primitive recursive schemes. So does the description of many aspects of dynamic semantics of languages.

Related Work

As far as we know, the only paper in which some kind of incremental term rewriting is mentioned is [FGJM85]. In that paper the term rewriting engine of OBJ2 is briefly described. OBJ2 is a functional programming language based upon equational logic and implemented as a term rewriting system. The computation of terms with a top symbol that has the "saveruns" annotation (set by the user) are stored in a hash-table. Before a term is computed this table is checked.

Attali and Franchi-Zannettacci [AFZ88, Att88] translate TYPOL programs that belong to a certain subclass completely to attribute grammars, in order to obtain partial and incremental evaluation of TYPOL programs. TYPOL is a formalism for specifying the semantics of programming languages based on Natural Semantics.

In [RTD83] Reps, Teitelbaum and Demers describe incremental attribute updating after a subtree replacement.

References

- [AFZ88] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Logic Programming Series, pages 160–177. MIT Press, 1988.
- [Att88] I. Attali. Compiling TYPOL with attribute grammars. In P. Deransart, B. Lorho, and J. Małuszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming '88*, volume 348 of *Lecture Notes in Computer Science*, pages 252–272. Springer-Verlag, 1988.
- [BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
- [CFZ82a] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I. *Theoretical Computer Science*, 17:163–191, 1982.
- [CFZ82b] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes II. *Theoretical Computer Science*, 17:235–257, 1982.
- [FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.
- [Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.
- [RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

Languages Polynomial In The Input Plus Output

Jiazhen Cai¹ and Robert Paige²

New York University/ Courant Institute

New York, NY 10012

and

University of Wisconsin

Madison, WI 53706

1. Introduction

The great gap between high level specifications of combinatorial problems and their low level implementations makes it hard to come up with software development tools or even systematic methods for transforming such formal specifications into efficient algorithm implementations. However, this gap is bridged in specific software generators such as YACC[7], where abstract specifications in the form of LALR(1) grammars are turned into linear time LR(k) parsers [8], and in symbolic algebra systems such as MACSYMA [10] with their broad practical scope and problem solving capabilities.

In [3] we attempted to extend some of the ideas underlying YACC and MACSYMA to a general purpose software generating language L_{IO} for specifying computable functions translatable into RAM implementations with worst case running time and auxiliary space linear in the input/output space. L_{IO} is a recursive subset of SQ2+ [4], a Turing-complete, functional problem specification language based on a computational finite set theory augmented with least and greatest fixed point expressions. L_{IO} is built up algebraically by restricted composition and parameter substitution from languages characterizing unit worst case and linear amortized time. Like MACSYMA L_{IO} sacrifices completeness (proved unattainable by Gurevich and Shelah unless partial functions are allowed [6]) in favor of programming convenience and reasonably wide ranging application. For example, in L_{IO} we can write perspicuous specifications of graph reachability, cycle testing, relational attribute closure, live code analysis, graph interval finding, finding all nonterminals in a CFG that derive the empty string, and constant propagation. We can also specify a significant fragment of an optimizing compiler. We implemented a running prototype compiler for L_{IO} .

This paper extends the earlier work on L_{IO} in the following ways:

- i. For each integer $k \geq 1$, we design an upper bound language L_k and a compiler that can translate any specification in L_k into an implementation with worst case running time and auxiliary space bounded by a k^{th} degree polynomial in the space needed to store the input plus output. These languages are recursive subsets of SQ2+. The construction makes use of algebraically defined languages approximating undecidable properties, such as 'monotone' and 'finite'.
- ii. In order to model nondeterministic problems we augment these languages with minimal and maximal fixed points of relations.

1. The research of this author was partially supported by National Science Foundation grant CCR-9002428.

2. The research of this author was partially supported by Office of Naval Research Grant No. N00014-90-J-1890.

- iii. Separate languages are delineated for three sequential models of computation: set machine (characterized by unit time associative access), array RAM (with unit time array access), and pointer RAM (with unit time pointer access and no pointer arithmetic). These languages are defined algebraically to satisfy sufficient conditions ensuring real-time simulation of an abstract model of computation on an array or pointer RAM. The algebraic formalism improves the ad hoc approach used earlier to guarantee real-time simulation for the RAM version of L_{110} .

2. Elementary Programs and Static Complexity

Because of space restrictions, we will only discuss the set machine (SM) model of computation, and will leave the discussion of real-time simulation of SM on a RAM to the long form of this paper. The set machine is distinguished by primitive operations on finite sets and maps (defined as sets of ordered pairs). Its primitives include unit-time operations for retrieval; e.g. $\ni S$ retrieves an arbitrary element of a set S , where $\ni \emptyset$ is undefined; control structure for $x \in S$ loop... searches through set S in $O(|S|)$ time by performing successive unit-time operations to retrieve the 'next' value. Adding an element x to a finite set S , denoted by $S \text{ with } := x$, also takes unit time, as does set initialization and re-initialization, denoted by $S := \emptyset$. Unit-time associative access is also supported; e.g. membership tests $x \in S$. If f is a finite function, then function application $f(x)$ is performed by a unit-time associative access that locates x in the domain of f followed by a unit-time retrieval from the range. If f is a finite multi-valued relation, then $f\{x\}$ is performed by an associative access on the domain of f and a unit-time retrieval to locate the image set $\{y: [x, y] \in f\}$. Conventional control (such as while-loops) and assignments are also supported.

The SM model serves two purposes. First, it is used to conveniently analyze costs of programs in an abstract functional language. Second, it is used as an intermediate language to be simulated in real-time on two conventional RAM models (for which see [5, 11]).

We consider a high level language with the following elementary programs, each SM-implementable in linear time with respect to the input and output size: range f , cardinality $|S|$, image of set S under map F (i.e. $f[S]$), set former $\{x \in S \mid K(x)\}$, where boolean valued qualifier $K(x)$ takes unit time, $S \cap T$, $S - T$, $S \cup T$, boolean valued quantifiers $\exists x \in S \mid K(x)$ and $\forall x \in S \mid K(x)$ when $K(x)$ takes unit time, cross product $S \times T$, map inverse f^{-1} , and map composition $f \circ g$ when either f , g , or one of their inverses is one-to-one.

The preceding elementary programs all belong to L_1 . However, L_1 (and L_k $k > 1$) is not closed under composition; e.g. cardinality $|T|$ and cartesian product $S \times Q$ for finite sets belong to L_1 , but their composition $|S \times Q|$ does not; it belongs to L_2 because of intermediate swell. More generally, for any $k = 2, \dots$ and $i < k$ expression $\big| \times_{i=1}^k S \big|$ belongs to L_k but not L_i . Consequently the construction of our language hierarchy makes use of composition restricted by program properties relating input and output space.

Consider expression

$$(1) \quad C = f(x_1, \dots, x_n)$$

with inputs x_i , $i = 1, \dots, n$ and output C . Expression (1) *absorbs* input x_i to the k^{th} degree if $\text{space}(x_i) = O(\text{space}(C)^k)$. It is *output bound* to the k^{th} degree if it absorbs each of its input parameters to the k^{th} degree. It is *input bound* to the k^{th} degree if $\text{space}(C) = O(\sum_{i=1}^n \text{space}(x_i)^k)$.

The preceding definitions lead to rules for building up simple complexity bound languages, starting from elementary programs.

THEOREM 1. (Composition) If $f(x_1, x_2, \dots, x_i)$ is in L_m and $g(y_1, y_2, \dots, y_r)$ is in L_n , then $h(x_1, x_2, \dots, x_i) = f(x_1, \dots, x_{i-1}, g(y_1, \dots, y_r), x_i, \dots, x_r)$ is in $L_{\max(km, kn)}$ whenever g is input bound to the k^{th} degree or f absorbs its i^{th} parameter to the k^{th} degree. If f is input-bound (respectively absorbs parameter x_i) to the m^{th} degree, and g is

input-bound (respectively absorbs parameter y_j) to the n^{th} degree, then h is input bound (respectively absorbs parameter y_j) to the mn^{th} degree.

It is important to observe that we are talking about complexity of programs running on machine SM and not mathematical functions. Therefore, it makes sense to say that $\exists S \times T$ is in L_2 but not L_1 , whereas $[\exists S, \exists T]$ is in L_1 even though both programs compute the same function. Because arbitrary selection does not absorb its input to any degree, composing it with any output bound expression is expensive. However, this operation exemplifies a whole range of operations whose costs can be improved by what Bird calls 'promotion' [1] (see also Burstall and Darling-ton [2]), which amounts to moving operations from outermost to innermost subexpression; e.g.

$\exists S \cup T \Rightarrow$ if $S \neq \emptyset$ then $\exists S$ else $\exists T$ endif

$\exists F[S] \Rightarrow$ if $\exists x \in \text{domain } F \mid x \in S$ then $F\{x\}$ else \emptyset endif

Such transformations are especially crucial to our treatment of nondeterministic programs in the next section.

3. Nondeterministic Fixed Point Programs and Dynamic Complexity

We model nondeterministic programs using arbitrary selection \exists . In our nondeterministic language an expression e with a free variable x is called an \exists -function of x , and the set of all possible values that e can have is denoted by $\text{All}(e)$. If e contains no occurrences of \exists , then it is called a function of x , in which case $\text{ALL}(e)$ contains no more than one value.

Let $f(x)$ be an \exists -function defined on some partially ordered set $(D, <, 0)$ satisfying a finite ascending chain condition (FACC). We say f is *inflationary* if $y \geq x$ for all $x \in D$ and $y \in \text{ALL}(f(x))$. An element $d \in D$ is a *fixed point* of f if $d \in \text{ALL}(f(d))$. A *least fixed point* (lfp) d of f , denoted by $\text{lfp } x.f$, is a fixed point of f such that for all $d' < d$, d' is not a fixed point of f . An *inductive fixed point* (ifp) d of f , denoted by $\text{ifp } x.f$, is a fixed point of f such that $d \in \text{ALL}(f^k(0))$ for some $k \geq 0$.

PROPOSITION 1. Let $f(x) = x \cup g(x)$ be a set valued \exists -function defined on some partially ordered set $(D, <, 0)$ satisfying FACC. Then there is some $k \geq 0$ such that $\text{ALL}(f^k(0))$ contains an ifp of f . In this case, an ifp of f can be computed by the following program:

```
(1)  $s := \emptyset$ 
    (while  $\exists x \in g(s) - s$ )
       $s \text{ with} := x$ ;
    end;
```

■

EXAMPLE 1. Let $D = \{0, 1, 2, 3, 4\}$ with the numerical ordering \leq , let E be a set of edges $\{[0, 2], [0, 3], [3, 4], [1, 1], [2, 2], [4, 4]\}$. For any $d \in D$, let $f(d) = \exists E\{d\}$. Then 1 is a lfp of f , but not an ifp; 2 and 4 are ifps, but not a lfp. ■

For any \exists -function f on D , we define a relation $R_f = \{[x, y] : x \in D, y \in \text{ALL}(f(x))\}$. Let R be any binary relation on D . We define an \exists -function $f_R(x) = \text{if } R\{x\} = \{\} \text{ then } x \text{ else } \exists(R\{x\})$. Also define $R^+ = R \cup \{[x, x] : x \text{ in } D \mid R\{x\} = \{\}\}$. Then we have

PROPOSITION 2. $R^+ = R_{f_R}$. If $M = \{x \in D \mid x \in R^+\{x\}\}$, then an element d in D is an ifp of f_R iff d in M . ■

Unfortunately, the definitions of the preceding section do not help us analyze expressions $\text{ifp } x.f$ in a reasonable way. In order to exploit the repeated calculation of $f(x)$ occurring in the computation (1) of $\text{ifp } x.f$, we develop a syntactic notion of dynamic complexity to bound these costs.

Suppose that invariant $C \in \text{All}(f(x_1, \dots, x_n))$ holds at a program point p , but is falsified just after p by a modification dx_i to an input parameter x_i . Then any code that re-establishes this invariant by modifying C and local variables immediately before and after p is called *difference code* for C relative to dx . Expression C is said to be

strongly continuous to the k^{th} degree with respect to a set D of modifications to x_1, \dots, x_n if the cost of executing (predefined) difference code for C relative to each modification in D is

$$O\left(\sum_{i=1}^n \text{space}(x_i)^{k-1}\right)$$

Expression C is said to be weakly continuous to the k^{th} degree with respect to a set D of modifications to x_1, \dots, x_n if the preprocessing cost of initially computing $f(x_1, \dots, x_n)$ and storing its value in C plus the dynamic cost of executing difference code for C relative to m arbitrary (possibly different) modifications drawn from D is

$$O\left(m \sum_{i=1}^n \text{space}(x_i)^{k-1} + \text{space}(C_I)^k + \text{space}(C_F)^k + \sum_{i=1}^n \text{space}(x_i)^k\right)$$

in the worst case, where C_I is the initial value of C , C_F is the final value of C , and $x_i, i = 1, \dots, n$, are initial values. The definitions of strong and weak continuity are relative to predefined difference code. There are, for example, obvious difference code rules for which image set $f[S]$ is weakly continuous in the first degree with respect to modifications S with: $x, f\{w\}$ with: $z, f\{w\}$ less: z , but exhibits only second (and not first) degree weak continuity if S less: x is also allowed.

Weak continuity is a syntactic formulation of amortized complexity (see Chapter 1 of Tarjan's book [12]). We believe that its ability to accurately measure seemingly erratic costs contributes greatly to the expressive power of our low order languages. This new concept lets us analyze the performance of *ifp*.

THEOREM 2. Let $f(x) = x \cup g(x)$ be a set valued \Rightarrow function defined on some partially ordered set $(D, <, 0)$ satisfying FACC. Then *ifp* $x.f$ is in L_n if either $f(x)$ is weakly continuous to the n^{th} degree relative to element additions to x or f is in L_{n-1} .

The following theorem shows how to build up a language of weakly continuous expressions, which can be used to define a language hierarchy $L_i, i=1, \dots$ for SM.

THEOREM 3. If f is in L_n and is strongly continuous to the k^{th} degree relative to D , then f is weakly continuous to the $\max(n, k)^{\text{th}}$ degree relative to D . If $C1 = g(x)$ is weakly continuous to the k^{th} degree relative to $D1$, if $D2$ is the set of modifications to $E1$ in the difference code for $E1$ relative to $D1$, and if $E2 = f(E1)$ is weakly continuous to the i^{th} degree relative to $D2$, then $f \circ g$ is weakly continuous to degree $\max(i, k)$ relative to $D1$ whenever f absorbs its input to the l^{th} degree.

EXAMPLE 2. Maximal independent set problem.

Let $G = (E, V)$ be an undirected graph. We call a subset x of V a maximal independent set (*mis*) of G if for any $[u, v] \in E$, at most one of u and v can be in x , and for any $u \in V - x$, there is a vertex $v \in x$ such that $[u, v] \in E$. It is not difficult to see that x is a *mis* of G iff x is an *ifp* of the following inflationary function:

$$f(x) = x \cup \{ \exists (V - x - E[x]) \}$$

where we assume that $\{z\} = \{\}$ if z is undefined. Image set $E[x]$ is weakly continuous in the first degree wrt to the set element additions to x , and set difference and set union are strongly continuous wrt to both element addition to and deletion from either operand. Thus, by Theorem 3 $V - x - E[x]$ is weakly continuous to the first degree, so that *ifp* $x.f$ is in L_1 .

In this case, every *ifp* of f is also a *lfp* of f , but not vice versa. For example, if $E = \{(a, b), (b, c), (c, a)\}$, $V = \{a, b, c\}$. Then $\{b, c\}$ is a *lfp* of f , but not an independent set of G . ■

We also have two perspicuous L_1 specifications of topological sorting, where one compiles into Knuth's algorithm [9] and the other into Tarjan's [12]. A straightforward L_3 specification of the single source shortest path problem with nonnegative weights will be presented in the long form of this paper.

References

1. Bird, R., "The Promotion and Accumulation Strategies in Transformational Programming," *ACM TOPLAS*, vol. 6, no. 4, pp. 487-504, Oct., 1984.
2. Burstall, R. and Darlington, J., "A Transformation System for Developing Recursive Programs," *JACM*, vol. 24, no. 1, pp. 44-67, Jan., 1977.
3. Cai, J. and Paige, R., "Binding Performance at Language Design Time," in *ACM POPL*, pp. 85 - 97, Jan, 1987.
4. Cai, J. and Paige, R., "Program Derivation by Fixed Point Computation," *Science of Computer Programming*, vol. 11, pp. 197-261, 1988/89.
5. Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E., "Type Transformation and Data Structure Choice," in *Proc.TC2 Working Conf. on Constructing Programs from Specifications*, May, 1991.
6. Gurevich, Y. and Shelah, S., "Time Polynomial In Input Or Output," *J. Symb. Logic.*, vol. 54, no. 3, pp. 1083-1088, Sept. 1989.
7. Johnson, S., "YACC - yet another compiler compiler," AT&T Bell Laboratories, Murray Hill, N.J., 1975.
8. Knuth, D. E., "Semantics of Context-free Languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127-145, 1968.
9. Knuth, D. E., *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, Addison-Wesley, 1973.
10. Math Lab Group edition 9, *MACSYMA Reference Manual*, Cambridge, Mass., 1977.
11. Paige, R., "Real-time Simulation of a Set Machine on a RAM," in *ICCI '89*, ed. W. Koczkodaj, Computing and Information, Vol II, pp. 69-73, 1989.
12. Tarjan, R., *Data Structures and Network Algorithms*, SIAM, 1984.

POLYNOMIAL RELATORS

Extended Abstract

Roland C. Backhouse* Peter J. de Bruin[†] Paul Hoogendijk* Grant Malcolm[‡]
Ed Voermans* Jaap van der Woude[§]

February 20, 1991

Abstract

This paper reports ongoing research into a theory of datatypes based on the calculus of relations. A fundamental concept introduced here is the notion of “relator” which is an adaption of the categorical notion of functor. Axiomatisations of polynomial relators (that is relators built from the unit type and the disjoint sum and cartesian product relators) are given and their properties extensively catalogued. The effectiveness of the calculus is illustrated by the construction of several natural isomorphisms and natural simulations between relators.

*Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

[†]Department of Computer Science, Rijksuniversiteit Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands.

[‡]Computing Laboratory, Programming Research Group, Oxford University, 8-11 Keble Road, Oxford OX1 3QD, United Kingdom.

[§]CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

1 Introduction

Research is underway into a theory of datatypes based on the calculus of relations. A fundamental concept within the theory is that of “relator” which takes the place of the notion of “functor” in a category-theory inspired development of functional programming.

In order that our theory be of any use we need to ensure that we can indeed define some non-trivial relators. It is well accepted that any useful theory of types should contain as a bare minimum the three components: a unit type, a disjoint sum operator and a cartesian product operator. In this paper we axiomatise these three within the calculus of relations and catalogue their properties. Following established terminology (see for example Manes and Arbib [6]) we call the relators so obtained the polynomial relators.

The main goal of our endeavour is to develop a calculus admitting compact and clear derivation of programs. To this end emphasis is placed on the identification of morphisms associated with each relator and their so-called “fusion” and “naturality” properties. Fusion properties characterise the circumstances under which a composition of relations, one of which is a morphism, can be combined into just one morphism (or vice-versa when a morphism can be “defused” into a composition of relations one of which remains a morphism). Naturality properties offer an alternative way of expressing monotonicity and fusion properties of morphisms; from a programming point of view their significance emerges when one tackles the problem of constructing simulations of one relator by another or isomorphisms between relators. The effectiveness of the calculus is demonstrated by exhibiting several such constructions.

Because of space limitations almost all proofs have been omitted from this extended abstract as well as all discussion of continuity properties. The complete paper is available on request from the authors. A companion paper [3] provides more motivation for the development of the theory and takes it further into the domain of inductively-defined types.

2 The Algebraic Framework

In this section we summarise relational algebra. For pedagogic reasons we decompose the algebra into three layers with interfaces between the layers plus two special axioms, one discussed here and one in a later section. The algebra is standard within the bounds of “scientific freedom”.

2.1 Plat Calculus

Let \mathcal{A} be a set, the elements of which are to be called *specs*. On \mathcal{A} we impose the structure of a complete, completely distributive, complemented lattice

$$(\mathcal{A}, \sqcap, \sqcup, \neg, \sqcap\sqcap, \sqcup\sqcup)$$

where “ \sqcap ” and “ \sqcup ” are associative and idempotent, binary infix operators with unit elements “ $\sqcap\sqcap$ ” and “ $\sqcup\sqcup$ ”, respectively, and “ \neg ” is the unary prefix operator denoting complement (or negation). We call such a structure a *plat*, the “p” standing for power set and “lat” standing for lattice. Since the structure is so well known and well documented we shall assume a high degree of familiarity with it.

2.2 Composition

The second layer is the monoid structure for composition:

$$(\mathcal{A}, \circ, I)$$

where \circ is an associative binary infix operator with unit element I .

The interface between these two layers is: \circ is coordinatewise universally "cup-junctive". I.e. for $V, W \subseteq \mathcal{A}$,

$$(\sqcup V) \circ (\sqcup W) = \sqcup (P, Q : P \in V \wedge Q \in W : P \circ Q)$$

2.3 Reverse

The third layer is the "reverse structure",

$$(\mathcal{A}, \cup)$$

where " \cup " is a unary postfix operator such that it is its own inverse.

The interface with the first layer is that " \cup " is an isomorphism of plats. I.e. for all $P, Q \in \mathcal{A}$,

$$P \supseteq Q \equiv P \cup \supseteq Q \cup$$

Remark As a rule we shall write the names of unary functions as prefixes to their arguments. A partial justification for making an exception of " \cup " is that it commutes with " \neg ", thus permitting us to write the syntactically ambiguous " $\neg R \cup$ ". Later we shall see that " \cup " also commutes (by definition) with so-called "relators". The latter is the main reason for this choice of notation.

End of Remark

The interface with the second layer is that " \cup " is an isomorphism between the two monoid structures $(\mathcal{A}, ;, I)$ and (\mathcal{A}, \circ, I) with

$$R ; S = S \circ R$$

2.4 Operator precedence

Some remarks on operator precedence are necessary to enable the reader to parse our formulae. First, as always, operators in the metalanguage have lower precedence than operators in the object language. The principle meta-operators we use are equivalence (" \equiv "), implication (" \Rightarrow ") and follows-from (" \Leftarrow ") — these all having equal precedence —, together with conjunction (" \wedge ") and disjunction (" \vee ") — which have equal precedence higher than that of the other meta-operators. The precedence of the operators in the plat structure follows the same pattern. That is, " $=$ ", " \supseteq " and " \sqsubseteq " all have equal precedence; so do " \sqcup " and " \sqcap "; and, the former is lower than the latter. Composition (" \circ ") has a yet higher precedence than all of the operators mentioned thus far, whilst disjoint sum "+", cartesian product " \times " and their associated morphisms, junction " ∇ " and split " Δ " (all of which are introduced later) have the highest precedence of all the binary operators. Finally, all unary operators in the object language, whether prefix or postfix, have the same precedence which is the highest of all. Parentheses will be used to disambiguate expressions where this is necessary.

2.5 The RS and Rotation Rules

To the above axioms we now add an axiom that acts as an interface between all three layers.

The RS Rule

$$\neg Y \supseteq P \circ \neg X \circ Q \equiv X \supseteq P \cup \circ Y \circ Q \cup$$

The name "RS" is a mnemonic for "Rotation and Shunting". The "rotation rule" is obtained by making the substitutions $Y := R\cup$, $P := S$, $X := \neg T$ and $Q := I$ and simplifying using the properties of I , reverse and complement.

Rotation Rule

$$\neg R\cup \supseteq S \circ T \equiv \neg T\cup \supseteq R \circ S$$

Both these rules are variants on the so-called Schröder rule. (See, for example, [10] for historical references.)

3 Foundations

The purpose of this section is to build up a vocabulary for our later discussion of the properties of morphisms. In order to avoid possible confusion with existing terminology we make a complete reappraisal of what is meant by "type", "function", "type constructor" etc. Nevertheless, it should be emphasised that — with the important exception of the notion of "relator" — *the concepts defined in this section, and their properties, are amply documented in the mathematical literature and we make no claim to originality.*

3.1 Monotypes

We say that $\text{spec } A$ is a *monotype* iff $I \supseteq A$. Note that for monotypes A and B

$$(1) \quad A = I \sqcap A = A\cup = A \circ A$$

$$(2) \quad A \circ B = B \circ A = A \sqcap B$$

We need to refer to the "domain" and "co-domain" (or "range") of a spec. In order to avoid unhelpful operational interpretations we use the terms *left-domain* and *right-domain* instead. These are denoted by " $<$ " and " $>$ ", respectively, and defined by

$$(3) \quad R< \equiv I \sqcap (R \circ R\cup)$$

$$(4) \quad R> \equiv I \sqcap (R\cup \circ R)$$

Since all the operators involved in their definition are monotonic it follows that " $<$ " and " $>$ " are monotonic. Relational calculus (see the appendix) yields the following alternative definitions defining $R<$ and $R>$ as the smallest monotypes satisfying the equations in A , $A \circ R = R$ and $R \circ A = R$, respectively.

$$(5) \quad \forall(A : I \supseteq A : A \circ R = R \equiv A \supseteq R<)$$

$$(6) \quad \forall(A : I \supseteq A : R \circ A = R \equiv A \supseteq R>)$$

Note that once again we choose to use a postfix notation for function application. On this occasion, however, it is *not* the case that complement and " $<$ " (or " $>$ ") commute. That is $\neg(R<) \neq (\neg R)<$, in general. As we shall see, however, " $<$ " and " $>$ " do commute with relators and that is the reason for our choice.

We sometimes write

$$R \in S \sim T$$

as a synonym for

$$(7) \quad S \circ R = R = R \circ T$$

It is immediate from (5) and (6) that

$$(8) \quad R< \circ R = R = R \circ R>$$

3.2 Imps and Co-imps

In this subsection we define "imps" and "co-imps" as special classes of specs. In the relational model an "imp" is a function.

Definition 9

- (a) A spec f is said to be an *imp* if and only if $I \supseteq f \circ f^\cup$.
- (b) A spec f is said to be a *co-imp* if and only if f^\cup is an imp.
- (c) A spec is said to be a *bijection* if and only if it is both an imp and a co-imp.

□

We shall say that f is a bijection *between* A and B if it is a bijection and $f^< = A$ and $f^> = B$. Note that if this is the case then both A and B are monotypes and $A = f \circ f^\cup$ and $B = f^\cup \circ f$. The notation " $A \cong B$ " (read as A is *isomorphic* to B) signifies the existence of a bijection between A and B .

Theorem 10 Composition preserves imps, co-imps and bijections.

□

The intended interpretation is that an "imp" is an "imp"lementation. On the other hand, it is not the intention that all implementations are "imps". Apart from their interpretation imps have an important distributive property not enjoyed by arbitrary specs, namely:

Theorem 11 If f is an imp then, for all specs R and S ,

$$(R \sqcap S) \circ f = (R \circ f) \sqcap (S \circ f)$$

□

Dually we have:

Theorem 12 If f is a co-imp then, for all specs R and S ,

$$f \circ (R \sqcap S) = (f \circ R) \sqcap (f \circ S)$$

□

Monotypes are examples of bijections. More generally, the requirement of being a function is the requirement of being single-valued on some subset of \mathcal{U} , the so-called "domain" of the function. The domain and range are made explicit in the following.

Definition 13 For monotypes A and B we define the set $A \longleftarrow B$ by $f \in A \longleftarrow B$ whenever

- (a) $I \supseteq f \circ f^\cup$
- (b) $f^> = B$, and
- (c) $A \supseteq f^<$

The nomenclature " $f \in A \longleftarrow B$ " is verbalised by saying that " f is an imp to A from B ".

□

We should stress that the two set-forming operations " \sim " and " \leftarrow " do *not* form an essential part of our theory but are included in order that the reader may relate their existing knowledge of type structures to the present theory. In the sequel we shall often state properties of the domain-forming operations " $<$ " and " $>$ " and immediately transcribe them into properties of " \sim " and/or " \leftarrow ". We prefer the statements about the domains for two reasons: they offer a better separation of concerns and are thus calculationally more useful, and they can be stated with fewer dummies (and indeed in some cases with no dummies, although we don't go that far).

To avoid repeating assumptions and to assist the reader's understanding we continue to use the conventions that capital letters A, B, C, \dots at the beginning of the alphabet denote monotypes, small letters f, g, h, \dots denote imps or co-imps, and capital letters R, S, T, \dots at the end of the alphabet denote arbitrary specs.

Finally, let us remark that the unconventional direction of the arrow in the statement " $f \in A \leftarrow B$ " is entirely dictated by the choice to denote function application with the function name to the left of its argument. (We owe the suggestion to deviate from convention to Meertens [7].)

3.3 Relators

In categorical approaches to type theory a parallel is drawn between the notion of type constructor and the categorical notion of "functor", thereby emphasising that a type constructor is not just a function from types to types but also comes equipped with a function that maps arrows to arrows. For an informative account of this parallel see, for example, [6]. In this subsection we propose a modest extension to the notion of functor to which we give the name "relator".

Definition 14 A *relator* is a function, F , from specs to specs such that

- (a) $I \supseteq F.I$
- (b) $R \supseteq S \Rightarrow F.R \supseteq F.S$
- (c) $F.(R \circ S) = F.R \circ F.S$
- (d) $F.(R \cup) = (F.R) \cup$

□

In view of (14d) we take the liberty of writing simply " $F.R \cup$ " without parentheses, thus avoiding explicit use of the property.

The above ostensibly defines a *unary* relator but we also wish to allow it to serve as the definition of a relator mapping an m -ary *vector* of specs into an n -ary *vector* of specs, for some natural numbers m and n . (This is necessary in order to allow the theory to encompass what are variously called "mutually recursive type definitions" and "many-sorted algebras". More generally, there is no reason why " m " and " n " may not be some fixed but nevertheless arbitrary index sets. However, such a generalisation would complicate the current discussion more than we deem justified.) The mechanism by which we can do this is to assume that all the constants appearing in the definition (" $=$ ", " \supseteq ", " I ", " \circ " and " \cup ") are silently "lifted" to operate on vectors. For example, if F maps m -ary vectors into n -ary vectors, property (14c) would be written out in the form

$$(F.(R_1 \circ S_1, \dots, R_m \circ S_m))_j = (F.(R_1, \dots, R_m))_j \circ (F.(S_1, \dots, S_m))_j$$

for all j , $1 \leq j \leq n$, whereby the use of subscripts denotes projection of a vector onto one of its components. It is, however, just such clumsy expressions that we want to avoid.

There are two cases that we make particular use of. The first case has to do with taking fixed points of relators where an obvious requirement is that the arity of the domain vector of the relator is identical to that of its range vector. We call such a relator an *endorelator* in conformance with the terminology “endofunctor” used in category theory. The second case is when F maps a pair of (vectors of) specs into a (vector of) spec(s). We refer to such relators as *binary* relators and choose to denote them by infix operators. Thus, if \otimes denotes a binary relator, its defining properties would be spelt out as follows.

- (a) $I \supseteq I \otimes I$
- (b) $R \supseteq S \wedge U \supseteq V \Rightarrow R \otimes U \supseteq S \otimes V$
- (c) $(R \circ S) \otimes (U \circ V) = (R \otimes U) \circ (S \otimes V)$
- (d) $(R^\cup) \otimes (S^\cup) = (R \otimes S)^\cup$

The notational advantage of writing “ \cup ” as a postfix to its argument is, of course, lost in this case.

As already announced relators commute with the domain operators.

Theorem 15 If F is a relator then

- (a) $F.(R>) = (F.R)>$
- (b) $F.(R<) = (F.R)<$

□

In view of theorem 15 we write “ $F.R<$ ” and “ $F.R>$ ” without parentheses, again in order to avoid explicit mention of the properties.

The following theorem allows a comparison to be made with our definition of “relator” and the definition of “functor” (in the category of sets).

Theorem 16 If F is a relator then

- (a) A is a monotype $\Rightarrow F.A$ is a monotype
- (b) f is an imp $\Rightarrow F.f$ is an imp
- (c) f is a co-imp $\Rightarrow F.f$ is a co-imp
- (d) $f \in A \longleftarrow B \Rightarrow F.f \in F.A \longleftarrow F.B$
- (e) $R \in A \sim B \Rightarrow F.R \in F.A \sim F.B$

□

4 Natural Polymorphism

Any discussion of a theory of datatypes would be incomplete without a discussion of polymorphism. This is particularly true here because our theory is principally a theory of two sets of polymorphic functions — the relators and their associated morphisms. Relators are polymorphic in the sense that they may be applied to arbitrary specs irrespective of the domains of the argument spec. Such a statement is, however, somewhat banal since it says nothing about the mathematical nature of the claimed polymorphism. In this section we shall argue that relators are “naturally polymorphic”. The latter notion is an adaptation and extension of the notion of “natural transformation” in category theory; the definition that we use is based on the work of de Bruin [5] which work was anticipated by Reynolds [8].

4.1 Higher-Order Spec Algebras

Expressing the natural polymorphism of relators (and other functions or relations) requires the notion of higher-order spec algebra which we now define.

Let $\text{SPEC} = (\mathcal{A}, \sqsubseteq, I, \circ, \cup)$ be a spec-algebra. Then the algebra of binary relations on specs $\overline{\text{SPEC}}$ is defined to be $(\overline{\mathcal{A}}, \overline{\sqsubseteq}, \overline{I}, \overline{\circ}, \overline{\cup})$ where

$$\begin{aligned}\overline{\mathcal{A}} &= \mathcal{P}(\mathcal{A} \times \mathcal{A}) \\ \overline{\sqsubseteq} &= \sqsubseteq\end{aligned}$$

and, using the notation $x \langle R \rangle y$ instead of $(x, y) \in R$,

$$\begin{aligned}x \langle \overline{I} \rangle y &\equiv x = y \\ x \langle R \overline{\circ} S \rangle z &\equiv \exists(y :: x \langle R \rangle y \wedge y \langle S \rangle z) \\ x \langle R \overline{\cup} \rangle y &\equiv y \langle R \rangle x\end{aligned}$$

for all $R, S \in \overline{\mathcal{A}}$ and all x, y and $z \in \mathcal{A}$. With these definitions, $\overline{\text{SPEC}}$ is also a spec algebra. We call $\overline{\text{SPEC}}$ a *higher-order spec algebra*.

The imps of $\overline{\text{SPEC}}$ are (partial) functions to \mathcal{A} from \mathcal{A} . Specifically, the function f from \mathcal{A} to \mathcal{A} is identified with the relation f on $\mathcal{A} \times \mathcal{A}$ where

$$x \langle f \rangle y \equiv x = f.y$$

for all $x, y \in \mathcal{A}$. Examples of imps in $\overline{\text{SPEC}}$ are the relators of SPEC. Note that relators are *total imps*. I.e. for each relator F we have

$$F \overline{\cup} \overline{\circ} F \sqsubseteq \overline{I}$$

The monotypes of $\overline{\text{SPEC}}$ can be identified with the subsets of \mathcal{A} . That is, a binary relation \overline{A} in $\overline{\mathcal{A}}$ is a *monotype* if and only if there is an element A of $\mathcal{P}(\mathcal{A})$ such that

$$(17) \quad \forall(x, y :: x \langle \overline{A} \rangle y \equiv x = y \wedge x \in A)$$

The operators " \sim " and " \longleftarrow " were defined in section 3.2 as set-forming operators. Using (17) to identify monotypes of $\overline{\text{SPEC}}$ with subsets of \mathcal{A} , we may identify " \sim " and " \longleftarrow " with elements of $\overline{\mathcal{A}}$, specifically with binary relations on elements of \mathcal{A} that are subsets of the identity relation \overline{I} . To reinforce this identification we corrupt the normal usage of the belongs-to symbol " \in " by the following definition. For spec R and relation \overline{S} we define

$$R \in \overline{S} \equiv R \langle \overline{S} \rangle R$$

Of course, $\overline{\text{SPEC}}$ can itself serve as the basis for the construction of a second algebra of binary relations $\overline{\overline{\text{SPEC}}}$, and in this way one can construct an infinite hierarchy of spec algebras. The relators and morphism constructors of one algebra are then total imps in the next higher order algebra; similarly, the expressions " $A \sim B$ " and " $A \longleftarrow B$ " of one algebra may be identified with monotypes in the next higher order algebra. Maintaining the distinction between the levels has been one reason why we have continually distinguished between "specs" and "relations", and between "imps" and "functions".

In this section we define three more relations which we call the naturality operators. The operators will be used at various levels in the hierarchy of SPEC algebras but we do not bother to decorate the different uses with a bar to indicate the level of use. Similarly, we use the undecorated symbols " \longleftarrow ", " \sim ", " \circ ", " \cup " etc. at all levels in the hierarchy. The definitions

of the barred operators given earlier will be important to reducing statements at one level to statements at the next lower level. Their use is, of course, only permitted within higher-order algebras.

As an example of this overloading of notation and in order to provide a reference point for our later discussion let us note the following properties:

Theorem 18 Let F be a relator. Then, for all monotypes A and B ,

- (a) $F \circ (A \sim B) \in (F.A \sim F.B) \longleftarrow (A \sim B)$
- (b) $F \circ (A \longleftarrow B) \in (F.A \longleftarrow F.B) \longleftarrow (A \longleftarrow B)$

□

To understand these statements one must understand at what level each of the operators is being used. Theorem 18(a) is exemplary. Reintroducing the bar notation it states that

$$F \bar{\circ} (A \sim B) \bar{\in} (F.A \sim F.B) \bar{\longleftarrow} (A \sim B)$$

Thus all operators are higher-order but for the “ \sim ” operators. Note that $F \bar{\circ} (A \sim B)$ is the restriction of relator F to elements of $A \sim B$. A more conventional (but computationally less convenient) notation might be $F_{A \sim B}$ (or $F_{A,B}$) indicating that relators are families of functions indexed by pairs of monotypes. Statement (b) is interpreted similarly; all operators are higher-order but for the first, second and fourth occurrences of “ \longleftarrow ”. The property is, in part, just another way of stating theorem 16(c) but with one fewer bound variable. Property 18(b) may be verified in a similar way; it restates theorem 16(b).

4.2 The Naturality Operators

Saving one bound variable is hardly justification for such a spate of definitions. The motivation for presenting theorem 18 was to be able to compare it to theorem 21 below. First, yet three more definitions.

Definition 19 (The Naturality Operators) Let R and S be specs. Then we define the relations $R \bar{\leftarrow} S$, $R \bar{\rightarrow} S$ and $R \bar{\leftrightarrow} S$ by

- (a) $U \langle R \bar{\leftarrow} S \rangle V \equiv R \circ V \supseteq U \circ S$
 - (b) $U \langle R \bar{\rightarrow} S \rangle V \equiv R \circ V \subseteq U \circ S$
- and
- (c) $U \langle R \bar{\leftrightarrow} S \rangle V \equiv R \circ V = U \circ S$

□

The above definition of the $\bar{\leftarrow}$ operator was introduced in [1]; it is related by part (a) of the following theorem to definitions introduced variously by deBruin [5], Reynolds [8] and Wadler [11].

Theorem 20

- (a) If R and S are relations and f and g are total functions then

$$f \langle R \bar{\leftarrow} S \rangle g \equiv \forall(u, v :: f.u \langle R \rangle g.v \Leftarrow u \langle S \rangle v)$$
- (b) If R and S are relations and f^\vee and g^\vee are surjective functions then

$$f \langle R \rightsquigarrow S \rangle g \equiv \forall(u, v :: u \langle R \rangle v \Rightarrow f \circ u \langle S \rangle g \circ v)$$

(c) If R and S are relations and f and g are total, surjective bijections then

$$f \langle R \leftrightarrow S \rangle g \equiv \forall(u, v :: f \circ u \langle R \rangle g \circ v \equiv u \langle S \rangle v)$$

□

See [1] for the (straightforward) proof of part (a) of this theorem.

Several other more evident properties of these operators will be assumed in the sequel, an example being that \leftarrow is anti-monotonic in its second argument.

4.3 Naturality of Relators, Reverse and Composition

The reader is invited to compare the following theorem with theorem 18.

Theorem 21 (Naturality of Relators) If F is a relator then for all specs R and S

$$(a) \quad F \in (F.R \leftarrow F.S) \leftarrow (R \leftarrow S)$$

$$(b) \quad F \in (F.R \rightsquigarrow F.S) \rightsquigarrow (R \rightsquigarrow S)$$

$$(c) \quad F \in (F.R \leftrightarrow F.S) \leftrightarrow (R \leftrightarrow S)$$

□

Nowhere in this document do we hazard a definition of “natural polymorphism”. Theorem 21 does, however, express precisely what we intend by the informal statement that relators are “naturally polymorphic”. Similar theorems are proved later about the basic constituents of cartesian products and disjoint sums. In each case the theorem involves a universal quantification over specs, and it is in this sense that the spec in question is “polymorphic”. The adjective “naturally” is added to suggest the link with “natural transformation” in category theory and to avoid confusion of our notion of polymorphism with existing notions.

Note: composition and reverse are also naturally polymorphic. See [2] for details.

5 The Unit Type

The unit type corresponds to a set with only one element; not a particularly interesting type, but nevertheless useful as a building block for constructing more complex data structures. The theory presented so far doesn’t provide a vocabulary for talking about elements, only for talking about specs: this is not unintentional since a goal of our work has always been to minimise the incidence of point-wise arguments. In keeping with this goal, we adopt a rather abstract view of data types, and take a roundabout route to characterise the unit type.

5.1 The Cone Rule

We begin by postulating an axiom dubbed “the cone rule”. This axiom could equally well have been included in section 2. It has been included here because it is only within the axiomatisation of the unit type that we make any use of the rule. Elsewhere (e.g. [10]) the cone rule is called “Tarski’s rule.”

The Cone Rule

$$\top \circ R \circ \top = \top \equiv R \neq \perp$$

As a partial motivation for the cone rule we ask the reader to compare it with the following consequence of the RS rule.

Lemma 22 For all specs R , the following statements are all equivalent:

- (a) $\perp\!\!\!\perp = R$
- (b) $\perp\!\!\!\perp = R \circ \top\!\!\!\top$
- (c) $\perp\!\!\!\perp = \top\!\!\!\top \circ R$
- (d) $\perp\!\!\!\perp = \top\!\!\!\top \circ R \circ \top\!\!\!\top$
- (e) $\perp\!\!\!\perp = R_{<}$
- (f) $\perp\!\!\!\perp = R_{>}$

□

One consequence of the cone rule is that $\top\!\!\!\top$ and $\perp\!\!\!\perp$ are different. More significantly, by combining the cone rule and lemma 22, one sees that the spec $\top\!\!\!\top \circ R \circ \top\!\!\!\top$ is always either $\top\!\!\!\top$ or $\perp\!\!\!\perp$ whatever the value of spec R . We say that the function mapping R to $\top\!\!\!\top \circ R \circ \top\!\!\!\top$ is *boolean-valued*; the cone rule itself is an abstract and concise way of expressing the proposition that, considered as a set of pairs, spec R either contains no elements or contains at least one element.

Another consequence of the cone rule, that we mention for later use, is the following:

Lemma 23 $\perp\!\!\!\perp = R \circ \top\!\!\!\top \circ S \equiv \perp\!\!\!\perp = R \vee \perp\!\!\!\perp = S$

□

5.2 The Axioms

In order to capture the notion of a unit type we need to express a sort of dual to the cone rule, namely that there is a non-empty spec which, when viewed as a set of pairs, consists of at most one pair the two components of which are identical. Specifically, we posit the existence of a spec, denoted $\mathbf{1}$, such that

$$(24) \quad \perp\!\!\!\perp \neq \mathbf{1}$$

and

$$(25) \quad I \supseteq \mathbf{1} \circ \top\!\!\!\top \circ \mathbf{1}$$

There are several ways to convince oneself that axioms (24) and (25) are indeed what we seek. One is to interpret the axioms in the relational model; another is to explore the consequences of the axioms within the theory itself. We would not discourage the reader from doing the former, but prefer ourself to emphasise the latter. We verify, first, that the unit type is an “atomic” monotype (“atomic” to be defined shortly) and, second, that it is a “terminal object” in the sense of category theory. Finally, we summarise certain basic properties of the unit type.

5.3 An Atomic Monotype

Theorem 26 $\mathbf{1}$ is a monotype.

□

We now define an *atom* to be a spec R such that, for every spec X ,

$$R \supseteq X \Rightarrow \perp\!\!\!\perp = X \vee R = X$$

Clearly, $\perp\!\!\!\perp$ is an atom. In general, the relational interpretation of an atom is a set of pairs containing at most one element.

Theorem 27 $\mathbf{1}$ is an atom

□

Properties (24), (26) and (27) express, respectively, that $\mathbf{1}$ is non-empty, and is a monotype corresponding to a set containing at most one element.

5.4 Terminality

The abstractness in the definition of the unit type consists, in part, of the fact that the unit type characterises *any* one-element set (or, if you prefer, is modelled by any one-element set); the identity of the element is irrelevant. In category theory a unit type is characterised by the following so-called “terminality” property: for each set A , there is one, and only one, function — commonly denoted by $!_A$ — in $\mathbf{1} \leftarrow A$. Letting $!$ denote $\mathbf{1} \circ \Pi$, this characterisation of the unit type is mimicked in our theory by the following two consequences of axioms (24) and (25). For all monotypes A ,

$$(28) \quad !_A \in \mathbf{1} \leftarrow A$$

$$(29) \quad R \in \mathbf{1} \sim A \wedge R > = A \equiv R = ! \circ A$$

Thus the categorical function $!_A$ is rendered by the imp $! \circ A$.

Equivalent, more succinct, and more fundamental, renderings of (28) and (29) are

$$(30) \quad ! \text{ is an imp, and}$$

$$(31) \quad \mathbf{1} = \mathbf{1} \circ \Pi \circ \mathbf{1}$$

from which follows

$$(32) \quad \mathbf{1} \supseteq R < \equiv R = ! \circ R >$$

It is also clear from these properties that $\mathbf{1}$ is unique up to isomorphism: if $\mathbf{1}'$ is also a unit type then $\mathbf{1} \circ \Pi \circ \mathbf{1}'$ is a bijection between $\mathbf{1}$ and $\mathbf{1}'$.

5.5 A Summary of Basic Properties

The “foundations” that were laid in sections 3 and 4 were not without purpose. In this and later sections we shall continually ask a number of standard questions about the specs and/or operators that have been newly introduced, the questions falling under headings such as “left and right domains”, “imps and co-imps”, and “natural polymorphism”. Two such questions have already been answered for the unit type: it is a monotype and the spec $!$ is an imp. To these we might also add that the function from specs to specs that always returns $\mathbf{1}$ is a relator (because $\mathbf{1}$ is a monotype). This seemingly trivial remark will prove to be quite important. There are two “standard questions” yet to be answered: what are the left and right domains of $!$ and in what sense is it naturally polymorphic? Here is the answer to the first of these.

Theorem 33

- (a) $!< = 1$
 (b) $!> = I$
 \square

Verification of both of these is straightforward and is left to the reader. (For part (a) make use of (31). For part (b) make use of the cone rule.)

The final question in this list is answered by the following theorem.

Theorem 34

$! \in 1 \Leftrightarrow \pi$
 In particular, for all specs R ,
 $! \in 1 \Leftarrow R$
 \square

The second naturality property of $!$ above is much the weaker of the two but may have a more familiar appearance. It is derived from the type statement

$$! \circ A \in 1 \longleftarrow A$$

by omitting the restriction of the domain to monotype A (in effect considering the polymorphic imp rather than an instance of it), replacing A by an arbitrary spec R and replacing " \longleftarrow " by " \Leftarrow ". It is this that is often meant by saying that $!$ is "naturally polymorphic".

The unit type constitutes a building block for the construction of data types; we turn now to the mortar: cartesian product and disjoint sum.

6 Axioms for Cartesian Product and Disjoint Sum

In all systems that we know of, cartesian product and disjoint sum are duals of each other. (Disjoint sum is indeed often given the name "co-product".) In choosing an axiomatisation of the two concepts in a relational framework we have therefore striven for two sets of rules that are "dual" to each other in some clearly recognisable way. It is for this reason that we present the two sets of axioms together in this section. In subsequent sections we consider separately the consequences of the axioms for cartesian product and for disjoint sum before returning in the final section to consider natural isomorphisms between combinations of the two.

In choosing our axioms, we have, of course, been strongly influenced by our experience with set-theoretic presentations of the relational calculus, that being the model our axioms are intended to capture. Since our notation is somewhat unconventional we shall frequently refer to this model for motivation.

We begin by postulating the existence of four specs, for cartesian product the two projections \ll (pronounced "project left") and \gg (pronounced "project right") and for disjoint sum the two injections \hookrightarrow (pronounced "inject left") and \hookleftarrow (pronounced "inject right"). (Note the unconventional direction of the arrow heads. As an aid to memory, and motivation for this choice, we suggest that the reader bear in mind the diagram " $X \hookrightarrow X+Y \hookleftarrow Y$ ".) Further, experience leads us to introduce four binary operators on specs, for cartesian product \triangle (pronounced "split") and \times (pronounced "times"), and for disjoint sum ∇ (pronounced "junc") and $+$ (pronounced "plus"), defined in terms of the projection and injection specs as follows:

$$(35) \quad P \triangle Q = (\ll \circ P) \sqcap (\gg \circ Q)$$

$$(36) \quad P \nabla Q = (P \circ \hookrightarrow) \sqcup (Q \circ \hookrightarrow)$$

$$(37) \quad P \times Q = (P \circ \ll) \triangle (Q \circ \gg)$$

$$(38) \quad P + Q = (\hookrightarrow \circ P) \nabla (\hookrightarrow \circ Q)$$

The relational model that we envisage assumes that the universe is a term algebra formed by closing some base set under three operators: the binary operator mapping the pair of terms x, y to the term (x, y) , and two unary operators \hookrightarrow and \hookleftarrow mapping the term x to the terms $\hookrightarrow.x$ and $\hookleftarrow.x$, respectively. The interpretation of \ll and \gg is that they project a pair onto its left and right components. That is,

$$x \langle \ll \rangle (x, y)$$

$$y \langle \gg \rangle (x, y)$$

The four defined operators should be familiar from their interpretations which are

$$(x, y) \langle P \triangle Q \rangle z \equiv x \langle P \rangle z \wedge y \langle Q \rangle z$$

$$x \langle P \nabla Q \rangle y \equiv \exists(z :: y = \hookrightarrow.z \wedge x \langle P \rangle z) \\ \vee \exists(z :: y = \hookleftarrow.z \wedge x \langle Q \rangle z)$$

$$(u, v) \langle P \times Q \rangle (x, y) \equiv u \langle P \rangle x \wedge v \langle Q \rangle y$$

$$x \langle P + Q \rangle y \equiv \exists(u, v :: x = \hookrightarrow.u \wedge y = \hookrightarrow.v \wedge u \langle P \rangle v) \\ \vee \exists(u, v :: x = \hookleftarrow.u \wedge y = \hookleftarrow.v \wedge u \langle Q \rangle v)$$

Note that these are the *definitions* of the operators in higher-order SPEC algebras.

Our first axiom is that the injections are both imps.

$$(39) \quad I \supseteq (\hookrightarrow \circ \hookrightarrow) \sqcup (\hookleftarrow \circ \hookleftarrow)$$

The "dual" of this axiom that we propose is:

$$(40) \quad I \supseteq (\ll \circ \hookrightarrow) \sqcap (\gg \circ \hookleftarrow)$$

which says that projecting a pair onto its first and second components and then recombining the components leaves the pair unchanged.

(Berghammer and Zierer [4] and de Roever [9] introduce an almost identical axiom to (40) but in their case the axiom is an equality rather than an inclusion. The difference is that their theories are monomorphic and not polymorphic. Relations are assumed to be (externally) typed and there is a family of product operators indexed by pairs of types. In our theory types (or rather domains) are internal and there is just one (polymorphic) product operator.)

We remark that axioms (39) and (40) take the following form when rephrased in terms of the product and sum operations.

$$(41) \quad I \supseteq I + I$$

$$(42) \quad I \supseteq I \times I$$

This is reassuring since it is one step on the way to guaranteeing that $+$ and \times are binary relators.

Cartesian product and conjunction are closely related. Specifically, we have (in the set-theoretic interpretation of \times)

$$x \langle P \cap Q \rangle y \equiv (x, x) \langle P \times Q \rangle (y, y)$$

Abstracting from this property in order to find an axiom that has a pleasing syntactic shape we are led to the following axiom:

$$(43) \quad (P \triangle Q) \circ (R \triangle S) = (P \circ R) \cap (Q \circ S)$$

The dual axiom for disjoint sum is:

$$(44) \quad (P \nabla Q) \circ (R \nabla S) = (P \circ R) \sqcup (Q \circ S)$$

(The reader may wish to interpret these properties in the relational model to assure themselves of their validity.)

As a final axiom we postulate that left projection is possible if and only if right projection is possible:

$$(45) \quad \llcorner = \rceil$$

Property (45) is equivalent to

$$(46) \quad \top \circ \llcorner = \top \circ \rceil$$

Its dual is therefore the trivially true

$$\hookrightarrow \circ \bot\bot = \leftarrow \circ \bot\bot$$

There are thus no further axioms for disjoint sum.

The five properties (39), (40), (43), (44) and (45) are the sum total of our axiomatisation of cartesian product and disjoint sum.

In the following two sections we consider individually the consequences of the axioms for cartesian product and disjoint sum. The cap operator in the definition of split together with the fact that composition is not universally cap-junctive make the calculations with cartesian product somewhat harder than those with disjoint sum. For this reason we begin with cartesian product and allow ourselves the luxury of much greater brevity in the discussion of disjoint sum. It should be noted that the organisation of the calculations in the next two sections is intended to facilitate, above all, ease of reference. A consequence thereof is that the reader may spot ways of shortening our calculations by interchanging the order of presentation.

7 Properties of Cartesian Product

There is a major complicating factor in developing a *relational* rather than a *functional* theory of datatypes. It is not, however, a complication that we want to avoid or brush under the carpet since it is an inevitable consequence of the desire to face the issue of nondeterminism. The complication can be pinpointed to cartesian product. Consider, as a first example, the "doubling function", i.e. a function that constructs a pair from a singleton by simply copying its argument. This is the imp $I \triangle I$. Now consider the equation:

$$(I \triangle I) \circ R = R \triangle R$$

and let us interpret R as a *nondeterministic function*. The equation is then clearly invalid since on the left side some nondeterministically calculated value is copied whereas on the right side a pair is constructed by applying R twice; since that calculation is nondeterministic the two elements of the pair may differ. If, however, R is a true function (an imp according to our definition) the equation is valid, as can easily be proved. Clearly the difference lies in the fact that imps distribute backwards over the cap operator whereas that is not the case in general.

The ramifications of the lack of such a distributivity property are manifold. They can best be observed by comparing the theorems in this section with those in the next. In particular

the fusion properties in the subsection 7.1, the computation rules in subsection 7.2 and the terminality property in subsection 7.5 are significantly less tractable than their counterparts for disjoint sum.

Many of the theorems in this section go in pairs: one for left and one for right projection. In all cases we prove just one of the two.

7.1 Fusion Properties

Our first concern is whether or not the product operator (\times) is a relator. According to the definition of a relator there are four conditions that we must verify. The first condition is axiomatically true (see (42)). The second condition, the requirement that cartesian product be monotonic in both its arguments, is clear from its definition (it is a composition of monotonic functions). Also clear from the definition of cartesian product is that the reverse operator distributes over it. I.e.

$$(47) \quad (P \times Q)^{\circ} = P^{\circ} \times Q^{\circ}$$

It remains to show that composition distributes over cartesian product:

Theorem 48 (Product-Split and Product-Product Fusion)

- (a) $(P \times Q) \circ (R \triangle S) = (P \circ R) \triangle (Q \circ S)$
- (b) $(P \times Q) \circ (R \times S) = (P \circ R) \times (Q \circ S)$

□

Properties (48a) and (48b) are the first examples of many properties to which we give the name "fusion" property. In general, whenever we introduce a relator we seek its associated "morphism" operator (in the case of cartesian product this is split, and in the case of disjoint sum this is junc) and we investigate conditions under which two specs can be "fused" into the one morphism. (Typically, as in (48a) and (48b) one of the specs to be fused is itself a morphism.) Elsewhere [3] we discuss relators defined via fixed-points and observe a connection between morphism fusion and loop fusion. Note, however, that we do not always use the rules to "fuse" specs; just as often we use them to "defuse" a spec into component parts. The reader should not allow the one-way character of the name to prejudice their use of such rules.

Remark Our efforts to identify categories of properties to which we give compact names can never be wholly satisfactory because the categories are not distinct. Property 48(b), for instance, is both a fusion property — because a product is a particular form of morphism — and an abide law — composition and product abide with each other. *End of Remark*

Corollary 49 \times is a binary relator.

□

A fusion equality in which the split occurs to the left of the composition cannot be established in general. An inclusion does hold, however, and is not entirely useless. Two cases where an equality can be established (although not the only ones) are when one operand of the split has the form $S \circ \Pi$ for some S and when the right operand of the composition is an imp.

Theorem 50 (Split-Imp Fusion)

- (a) $(R \triangle S) \circ T \supseteq (R \circ T) \triangle (S \circ T)$
- (b) $(R \triangle (S \circ \Pi)) \circ T = (R \circ T) \triangle (S \circ \Pi)$

and, for allimps f ,

- (b) $(R \triangle S) \circ f = (R \circ f) \triangle (S \circ f)$

□

7.2 Computation Rules

The name "projection" immediately suggests its operational interpretation. Here that interpretation is represented by two rules that we call "computation rules".

Note that the rules are valid for all specs P and Q but the righthand side of each rule is slightly more complex than a naive examination might suggest.

Theorem 51 (Computation Rules for Split)

- (a) $\ll \circ (P \triangle Q) = P \circ Q >$
 - (b) $\gg \circ (P \triangle Q) = Q \circ P >$
-

Theorem 52 (Product is Strict)

$$R \times S = \perp\!\!\!\perp \equiv R = \perp\!\!\!\perp \vee S = \perp\!\!\!\perp$$

□

Theorem 53 (Computation Rules for Product)

- (a) $\ll \circ (P \times Q) = P \circ \ll \circ (I \times Q >)$
 - (b) $\gg \circ (P \times Q) = Q \circ \gg \circ (P > \times I)$
-

7.3 Imp and Co-imp Preservation

Up till now our language has implied that the projections areimps but, as yet, we have not stated the fact so explicitly and nor has it been proven. Not surprisingly the proof is rather trivial.

Lemma 54

$$\ll \circ \ll^\cup = I \text{ and } \gg \circ \gg^\cup = I$$

□

Corollary 55 \ll and \gg are both imps.

□

Theorem 56

- (a) $P \triangle Q$ is a co-imp $\Leftarrow P$ is a co-imp $\vee Q$ is a co-imp.
 - (b) $P \triangle Q$ is an imp $\Leftarrow P$ is an imp $\wedge Q$ is an imp.
-

Since product is a binary relator we have:

Theorem 57 \times preserves both imps and co-imps.

□

(The fact that a split is a co-imp if just one of its arguments is a co-imp does not help one to prove anything stronger about product.)

7.4 Left and Right Domains

Much of the work necessary to determine the effect of the left and right domain operators on splits and left and right projections has already been completed. Lemma 54, for instance, tells us immediately that the projections are surjective. I.e.

Theorem 58 $\llcorner = I$ and $\lrcorner = I$

□

Moreover,

Theorem 59 $\llcorner \rceil = I \times I$ and $\lrcorner \lrcorner = I \times I$

□

Since product is a (binary) relator we can immediately instantiate theorem 15 obtaining:

Theorem 60

$$(a) \quad (P \times Q) \lrcorner = P \lrcorner \times Q \lrcorner$$

$$(b) \quad (P \times Q) \rceil = P \rceil \times Q \rceil$$

□

We conclude this section with expressions for the right and left domains of a split. That for the left domain is not particularly helpful but is more compact than the expanded form of the definition! (As one might expect it is usually more difficult to predict the left domain than the right domain of a split.)

Theorem 61 (Split Right and Left Domain)

$$(a) \quad (P \triangle Q) \rceil = P \rceil \sqcap Q \rceil$$

$$(b) \quad (P \triangle Q) \lrcorner = \llcorner \circ P \circ Q \circ \lrcorner \sqcap I$$

□

7.5 Unique Extension Properties

In the category **Set** of sets and total functions cartesian product is defined via limits of functors in the following way. Let **2** be the discrete category with objects $\{0,1\}$. For object A in **Set** the constant A functor is denoted by $\underline{A} : \mathbf{Set} \leftarrow \mathbf{2}$. The cartesian product of X and Y is defined to be the limit of the functor $\mathcal{F} : \mathbf{Set} \leftarrow \mathbf{2}$ with $\mathcal{F} \cdot 0 = X$ and $\mathcal{F} \cdot 1 = Y$. I.e. the product is a set C and a natural transformation $\pi : \mathcal{F} \leftarrow \underline{C}$ such that for every set D and every natural transformation $\sigma : \mathcal{F} \leftarrow \underline{D}$ there is a unique arrow $\phi : C \leftarrow D$ in **Set** such that $\pi \circ \phi = \sigma$. Usually C is denoted by $X \amalg Y$ or $X \times Y$, while the natural transformation is denoted by the pair π_X, π_Y of projections. The terminality of π is most often phrased as follows. For every D and all total functions $f : X \leftarrow D$ and $g : Y \leftarrow D$ there is a unique total function $h : X \amalg Y \leftarrow D$ such that $\pi_X \circ h = f$ and $\pi_Y \circ h = g$. In our system this terminality is valid not only for **imps** (our equivalent of functions) but also for a more general class of **specs** (although not for all **specs**). We refer to the relevant theorem as the "unique extension property" for cartesian product, and it is the purpose of this section to present the property and then to explore some of its consequences. First, a useful lemma.

Lemma 62

$$P \triangle Q = (P \circ Q >) \triangle (Q \circ P >)$$

□

In its most general form the unique extension property is as follows:

Theorem 63 (Unique Extension Property)

Suppose $\triangle \in \{\sqsubseteq, =, \sqsupseteq\}$. Assume also that

$$\llcirc \circ \llcirc \circ X \sqcap \ggcirc \circ \ggcirc \circ X \triangle X$$

Then

$$X \triangle P \triangle Q \equiv \llcirc \circ X \triangle P \circ Q > \wedge \ggcirc \circ X \triangle Q \circ P >$$

□

More often than not we apply the theorem with the variable " \triangle " instantiated to " $=$ ". However, since our purpose is to develop a theory that admits program refinement as a possible step we are continually on the lookout for more general properties of the same nature as theorem 63, the cost in terms of burden of proof being typically almost negligible.

The assumption in theorem 63 is somewhat unwieldy; however, it is important to note that it is *not* equivalent to X being an imp but it is only implied by that circumstance (for all instantiations of \triangle). The assumption is indeed quite weak and we shall encounter several instances where it is valid. One such case is where X is itself a split term, resulting in the following cancellation property.

Theorem 64 (Split Cancellation)

For all $\triangle \in \{\sqsubseteq, =, \sqsupseteq\}$

$$P \triangle Q \triangle R \triangle S \equiv P \circ Q > \triangle R \circ S > \wedge Q \circ P > \triangle S \circ R >$$

□

We return now to the original concern, which was the case that X , P and Q are all imps. The backwards distribution of imps over intersection shows that the assumption in the statement of the unique extension property is met for imps with left domain in $I \times I$. For the terminality we also have to get rid of the right domains. This explains the assumptions in the terminality theorem.

Theorem 65 (Terminality)

Let f be an imp with $I \times I \sqsupseteq f <$ and let $P > = Q >$. Then

$$f = P \triangle Q \equiv \llcirc \circ f = P \wedge \ggcirc \circ f = Q$$

Equivalently, for all imps f and all specs P, Q ,

$$(I \times I) \circ f = P' \triangle Q' \equiv \llcirc \circ f = P' \wedge \ggcirc \circ f = Q'$$

where P' denotes $P \circ Q >$ and Q' denotes $Q \circ P >$.

□

7.6 Naturality Properties

Part (a) of lemma 48 is a very important property, just as important as part (b). It can be expressed somewhat differently, namely as a naturality property of split.

Theorem 66 (Naturality of Split)

$$\Delta \in (R \times S \dot{\hookrightarrow} T) \dot{\hookrightarrow} (R \dot{\hookrightarrow} T) \times (S \dot{\hookrightarrow} T)$$

□

Note: it is not the case that any of the “ $\dot{\hookrightarrow}$ ” operators can be replaced by either “ $\dot{\hookrightarrow}$ ” or “ $\dot{\hookrightarrow}$ ”.

Since product is a (binary) relator we can simply instantiate theorem 21 to obtain:

Theorem 67 (Naturality of Product)

For all specs R, S, T, U and all $\sim \in \{\dot{\hookrightarrow}, \dot{\hookrightarrow}, \dot{\hookrightarrow}\}$

$$\times \in (R \times T \sim S \times U) \dot{\hookrightarrow} (R \sim S) \times (T \sim U)$$

□

The two projections are also naturally polymorphic in the following sense.

Theorem 68 (Naturality of Left and Right Projection)

For all specs R ,

$$(a) \quad \ll \in R \dot{\hookrightarrow} R \times \Pi \quad \text{and} \quad \gg \in R \dot{\hookrightarrow} \Pi \times R$$

In particular, for all specs R and S ,

$$(b) \quad \ll \in R \dot{\hookrightarrow} R \times S \quad \text{and} \quad \gg \in R \dot{\hookrightarrow} S \times R$$

□

(Note that “ $\dot{\hookrightarrow}$ ” in the statement of the theorem can be replaced by “ $\dot{\hookrightarrow}$ ” or “ $\dot{\hookrightarrow}$ ” since equality implies inclusion.)

8 Properties of Disjoint Sum

We have discussed the properties of cartesian product before those of disjoint sum because the latter are substantially simpler to derive. This is because the cap operator in the definition of split is replaced by the cup operator in the definition of junc, and composition is universally cup-junctive but not universally cap-junctive. Calculations with split and/or the projections can thus often be transliterated into calculations with junc and/or the injections — but less often the other way round. The order of presentation remains the same so that the reader may compare the properties one-by-one.

8.1 Fusion Properties

As was the case for cartesian product it is straightforward to see that $+$ satisfies three of the conditions necessary for it to be a relator: the first is satisfied axiomatically, and monotonicity and commutation with reverse are satisfied by construction. Distributivity with respect to composition is also a special case of a “fusion” law, namely that a sum can be fused with a junc.

Theorem 69 (Junc-Sum and Sum-Sum Fusion)

$$(a) \quad (P \nabla Q) \circ (R+S) = (P \circ R) \nabla (Q \circ S)$$

$$(b) \quad (P+Q) \circ (R+S) = (P \circ R) + (Q \circ S)$$

□

Corollary 70 $+$ is a relator.

□

One more fusion property can be added to this list on account of the universal cup-junctivity of composition, namely:

Theorem 71 (Spec-Junc Fusion)

$$P \circ (Q \nabla R) = (P \circ Q) \nabla (P \circ R)$$

□

Comparison should be made with theorem 50 where a restriction to imps had to be made in order to obtain an equality.

8.2 Computation Rules

For want of inventiveness we give the name "co-strictness" to the dual of the strictness of product.

Theorem 72 (Co-strictness of Sum)

$$R+S = \perp\perp \equiv R = \perp\perp \wedge S = \perp\perp$$

□

The computation rules for junc do not involve any extra complications (unlike those for split). Their derivation, however, follows the same pattern.

Theorem 73 (Computation Rules for Junc and Sum)

$$(a) \quad (P \nabla Q) \circ \hookrightarrow = P$$

$$(b) \quad (P \nabla Q) \circ \hookleftarrow = Q$$

In particular

$$(c) \quad (P+Q) \circ \hookrightarrow = \hookrightarrow \circ P$$

$$(d) \quad (P+Q) \circ \hookleftarrow = \hookleftarrow \circ Q$$

□

8.3 Imp and Co-imp Preservation

Our first axiom was that left and right injection are both imps. In fact they are also co-imps as is evidenced by the following:

Lemma 74

$$\hookrightarrow \cup \circ \hookrightarrow = I = \hookleftarrow \cup \circ \hookleftarrow$$

□

Corollary 75 \hookrightarrow and \hookleftarrow are bijections.

□

Since split preserves bothimps and co-imps one would expect that junc does so too. But this is not the case! The proof that split preserves co-imps cannot be transliterated into a proof that junc preserves co-imps (thus emphasising that one has to be very careful with “dualisation” of arguments) and we can only assert that it preservesimps. Nevertheless, $+$ preserves both.

Theorem 76 (Imp and Co-imp Preservation)

- (a) ∇ preservesimps.
- (b) If f and g are both co-imps and $f < \sqcap g < = \perp\perp$ then $f \nabla g$ is a co-imp.
- (c) $+$ preserves bothimps and co-imps.

□

8.4 Left and Right Domains

Lemma 74 not only predicts that the injections are co-imps but also that they are total. Formulae for the left domain of the injections are also easy to calculate:

Theorem 77

- (a) $\hookrightarrow > = I$ and $\hookleftarrow > = I$
- (b) $\hookrightarrow < = I + \perp\perp$ and $\hookleftarrow < = \perp\perp + I$

□

The next theorem could be said to be the dual to the theorem that the right domains of the projections are equal.

Theorem 78

$$\hookrightarrow < \sqcap \hookleftarrow < = \perp\perp$$

□

In contrast to those for cartesian product the rules for the left and right domains of junc and sum are very simple. Both domain operators distribute over sum, and over junc, but transforming the operator in one case into cup and in the other into sum.

Theorem 79

- (a) $(P+Q) > = P > + Q >$
- (b) $(P+Q) < = P < + Q <$
- (c) $(P \nabla Q) < = P < \sqcup Q <$
- (d) $(P \nabla Q) > = P > + Q >$

□

8.5 Unique Extension Property

The counterpart of the *terminality* property of cartesian product is an *initiality* property. Here it is yet stronger: so much so indeed that it warrants a different order of presentation. The key insight is that two components in a junc or sum remain truly disjoint. To be precise:

Theorem 80 (Cancellation Properties)

For all $\trianglelefteq \in \{\sqsubseteq, =, \sqsupseteq\}$,

$$(a) \quad P \nabla Q \trianglelefteq R \nabla S \equiv P \trianglelefteq R \wedge Q \trianglelefteq S$$

$$(b) \quad P + Q \trianglelefteq R + S \equiv P \trianglelefteq R \wedge Q \trianglelefteq S$$

□

Corollary 81 (Junc Initiality)

For all $\trianglelefteq \in \{\sqsubseteq, =, \sqsupseteq\}$,

$$P \circ (I + I) \trianglelefteq Q \nabla R \equiv P \circ \hookrightarrow \trianglelefteq Q \wedge P \circ \hookleftarrow \trianglelefteq R$$

□

8.6 Naturality Properties

The naturality properties of the two injections are stronger than those of the projections.

Theorem 82 (Naturality of Left and Right Injection)

For all specs R and S ,

$$(a) \quad \hookrightarrow \in R + S \leftrightarrow R$$

$$(b) \quad \hookleftarrow \in R + S \leftrightarrow S$$

□

The naturality property of junc is also stronger.

Theorem 83 (Naturality of Junc and Sum)

For all specs R, S, T and U and all $\sim \in \{\hookleftarrow, \hookrightarrow, \leftrightarrow\}$,

$$(a) \quad \nabla \in (R \sim S + T) \hookleftarrow (R \sim S) \times (R \sim T)$$

$$(b) \quad + \in (R + S \sim T + U) \hookleftarrow (R \sim T) \times (S \sim U)$$

□

9 Natural Simulations and Natural Isomorphisms

To summarise, we now have one non-trivial monotype and two binary relators. Unary relators can be derived from these by fixing one of the arguments to a monotype; ternary relators, quaternary relators etc. can be obtained by composing them in appropriate ways; and new monotypes can be obtained by applying existing relators to existing monotypes. For example, $1+1$ and $1 \times (1+1)$ are monotypes, and the functions $1+$ and $(1 \times 1)+$ are unary relators. Relators and monotypes built in this way are called *polynomial*. This, however, is just the foundation. It is only now that our theory can really begin.

In this section we make a modest start to showing the ease with which certain calculations can be made within the theory. At the same time we formulate a number (at this point in time 2!) of concepts of particular relevance to data reification.

Definition 84 (Natural Simulation) Relator F is said to (naturally) simulate relator G if and only if there exists a spec γ such that

- (a) for all specs R , $\gamma \in F.R \leftrightarrow G.R$
- (b) γ is an imp
- (c) $\gamma> = G.I$

The spec γ itself is called a natural simulation.

□

Note that the combination of conditions (a) and (b) implies that $\gamma< \subseteq F.I$.

Definition 85 (Natural Isomorphism) Relators F and G are said to be naturally isomorphic if and only if there exists a spec γ such that

- (a) for all specs R , $\gamma \in F.R \leftrightarrow G.R$
- (b) γ is a bijection
- (c) $\gamma< = F.I$ and $\gamma> = G.I$

The spec γ itself is called a natural isomorphism.

□

Examples of natural isomorphisms are provided by the two injections \hookrightarrow and \hookleftarrow . The former is a natural isomorphism between the relator $(+\perp\perp)$ and the identity relator. I.e.

$$\begin{aligned} \hookrightarrow &\in R+\perp\perp \leftrightarrow R, \text{ for all specs } R \\ \hookrightarrow &\text{ is a bijection, and} \\ \hookrightarrow< &= I+\perp\perp \quad \text{and} \quad \hookrightarrow> = I \end{aligned}$$

Similarly, the latter is an isomorphism between the the relator $(\perp\perp+)$ and the identity relator. The injections are also examples of natural simulations: \hookrightarrow is, for example, a natural simulation of the identity relator by the relator $+1$. (In general any monotype may be used in place of 1 .)

As might be expected, both natural simulations and natural isomorphisms enjoy many simple but powerful algebraic properties. In later versions of this paper it is our intention to document some of them. For the time being, however, we leave the reader the pleasure of their discovery. Let us proceed to more significant examples. We begin with the most complicated, basic example of a natural isomorphism.

Consider the ternary relators defined by

$$\begin{aligned} R, S, T &\mapsto R \times (S+T) \\ R, S, T &\mapsto (R \times S)+(R \times T) \end{aligned}$$

Our objective is to show that the two relators are naturally isomorphic.

To complete this task we must exhibit a spec, γ , satisfying three quite strong conditions. We can make progress in this task by temporarily setting aside two of the conditions, *constructing* γ to satisfy the remaining condition, and then (hopefully) *verifying* that it satisfies the two other conditions. The condition singled out should be the one that leaves the least freedom to manoeuvre, in this case clearly condition (a). What we shall now demonstrate is how systematically this can be done using the rules we have given for the naturally polymorphic type of the operators we have introduced.

Here then is the construction of the desired natural isomorphism. Assume R , S , and T are arbitrary specs. Then

by construction of γ :

$$\begin{aligned}
& \gamma \in R \times (S+T) \Leftrightarrow (R \times S) + (R \times T) \\
\Leftarrow & \quad \{ \text{naturality of } \nabla, \gamma := \gamma_1 \nabla \gamma_2 \} \\
& \gamma_1 \in R \times (S+T) \Leftrightarrow R \times S \\
& \wedge \gamma_2 \in R \times (S+T) \Leftrightarrow R \times T \\
\Leftarrow & \quad \{ \text{naturality of product, } I \in R \Leftrightarrow R, \\
& \quad \gamma_1 := I \times \gamma_1, \gamma_2 := I \times \gamma_2 \} \\
& \gamma_1 \in S+T \Leftrightarrow S \wedge \gamma_2 \in S+T \Leftrightarrow T \\
\Leftarrow & \quad \{ \text{naturality of the injections} \} \\
& \gamma_1 = \hookrightarrow \wedge \gamma_2 = \hookleftarrow
\end{aligned}$$

Thus the constructed spec is γ where

$$\gamma = (I \times \hookrightarrow) \nabla (I \times \hookleftarrow)$$

It remains to show that γ is a bijection and has the correct left and right domains. The verifications are straightforward, but we give them nonetheless as proof of the pudding.

First, we assert that γ is a bijection. That it is an imp follows because it is built out ofimps using imp-preserving operators. Since junc is not necessarily co-imp preserving we need to take further steps to show that it is a co-imp.

$$\begin{aligned}
& \gamma \text{ is a co-imp} \\
\Leftarrow & \quad \{ \text{theorem 76(b)} \} \\
& (I \times \hookrightarrow \text{ and } I \times \hookleftarrow \text{ are co-imps}) \\
& \wedge (I \times \hookrightarrow)^< \sqcap (I \times \hookleftarrow)^< = \perp\perp \\
\equiv & \quad \{ \text{theorem 56} \} \\
& (I \times \hookrightarrow)^< \sqcap (I \times \hookleftarrow)^< = \perp\perp \\
\equiv & \quad \{ \text{theorem 60} \} \\
& (I \times \hookrightarrow^<) \sqcap (I \times \hookleftarrow^<) = \perp\perp \\
\equiv & \quad \{ \text{distributivity property in full paper} \} \\
& I \times (\hookrightarrow^< \sqcap \hookleftarrow^<) = \perp\perp \\
\equiv & \quad \{ \text{theorem 78, product is strict} \} \\
& \text{true}
\end{aligned}$$

We now calculate the left domain of γ .

$$\begin{aligned}
& \gamma^< \\
= & \quad \{ \text{definition of } \gamma, \text{ theorem 79(c)} \} \\
& (I \times \hookrightarrow)^< \sqcup (I \times \hookleftarrow)^< \\
= & \quad \{ \text{theorems 60 and 77} \} \\
& I \times (I + \perp\perp) \sqcup I \times (\perp\perp + I) \\
= & \quad \{ \text{distributivity property in full paper} \} \\
& I \times ((I + \perp\perp) \sqcup (\perp\perp + I)) \\
= & \quad \{ \text{definition of } + \} \\
& I \times (I + I)
\end{aligned}$$

Finally, we calculate the right domain of γ .

$$\begin{aligned}
& \gamma^> \\
= & \quad \{ \text{definition of } \gamma, \text{ theorem 79(d)} \}
\end{aligned}$$

$$\begin{aligned}
& (I \times \hookrightarrow)^> + (I \times \hookleftarrow)^> \\
= & \{ \text{theorems 60 and 77} \} \\
& (I \times I) + (I \times I)
\end{aligned}$$

This completes the verification.

The point of discussing this example in so much detail is to emphasise the importance of type considerations in *constructing* specs having prescribed properties. (This is a somewhat different emphasis than that which one encounters most frequently. Wadler [11], for example, discusses the use of natural polymorphism to *infer* properties of already constructed functions.) There is, however, yet more that can be said about the bijection γ that we have constructed that so far as we know is not predicted by any naturality theorem and yet seems "obvious" from type considerations. The properties that we allude to record its behaviour with respect to the two morphisms *split* and *junc*. Before stating and proving the properties we need to interpose a truly remarkable and elegant law permitting an exchange of *split* for *junc* and vice-versa.

Theorem 86 (Split-Junc Abide Law)

$$(R \nabla S) \triangle (T \nabla U) = (R \triangle T) \nabla (S \triangle U)$$

Proof We aim to use the initiality property (theorem 81) of *junc*. First note that

$$\begin{aligned}
& (R \nabla S) \triangle (T \nabla U) \circ I+I \\
= & \{ I+I \text{ is a monotype and thus an imp, (50)} \} \\
& (R \nabla S \circ I+I) \triangle (T \nabla U \circ I+I) \\
= & \{ \text{split fusion (48)} \} \\
& (R \nabla S) \triangle (T \nabla U)
\end{aligned}$$

Hence:

$$\begin{aligned}
& (R \nabla S) \triangle (T \nabla U) = (R \triangle T) \nabla (S \triangle U) \\
\equiv & \{ \text{theorem 81 combined with the above} \} \\
& (R \nabla S) \triangle (T \nabla U) \circ \hookrightarrow = R \triangle T \\
\wedge & (R \nabla S) \triangle (T \nabla U) \circ \hookleftarrow = S \triangle U
\end{aligned}$$

Continuing now with just one of the conjuncts in the last expression we calculate:

$$\begin{aligned}
& (R \nabla S) \triangle (T \nabla U) \circ \hookrightarrow \\
= & \{ \hookrightarrow \text{ is an imp, split-imp fusion} \} \\
& (R \nabla S \circ \hookrightarrow) \triangle (T \nabla U \circ \hookrightarrow) \\
= & \{ \text{junc-computation} \} \\
& R \triangle T
\end{aligned}$$

The other conjunct being dealt with in a similar way our proof is now complete.
□

The properties of the natural isomorphism γ that we anticipated above can now be given.

Theorem 87

- (a) $\gamma \circ (R \triangle S) + (R \triangle T) = (R \circ I \nabla I) \triangle (S+T)$
 (b) $R \times (S \nabla T) \circ \gamma = (R \times S) \nabla (R \times T)$

Proof

- (a) $\gamma \circ (R \triangle S) + (R \triangle T)$
 = { definition of γ , junc fusion theorem 69(a) }
 $(I \times \hookrightarrow \circ R \triangle S) \nabla (I \times \hookrightarrow \circ R \triangle T)$
 = { split fusion theorem 48(a) }
 $(R \triangle (\hookrightarrow \circ S)) \nabla (R \triangle (\hookrightarrow \circ T))$
 = { abides law (86) }
 $(R \nabla R) \triangle ((\hookrightarrow \circ S) \nabla (\hookrightarrow \circ T))$
 = { junc fusion (69a), definition of sum (38) }
 $(R \circ I \nabla I) \triangle (S+T)$
- (b) $R \times (S \nabla T) \circ \gamma$
 = { definition of γ , spec-junc fusion (71) }
 $(R \times (S \nabla T) \circ I \times \hookrightarrow) \nabla (R \times (S \nabla T) \circ I \times \hookrightarrow)$
 = { \times is a relator, junc computation rules (73) }
 $(R \times S) \nabla (R \times T)$

□

Natural isomorphisms seem to receive scant attention in the category theory literature, often being relegated to a brief exercise. This is somewhat unfortunate because it deemphasises their importance and it means that no guidance is given on how to construct them. We also relegate the construction of several basic natural isomorphisms to the present set of exercises, not because they are unimportant but because by doing them the reader may be enabled to make a judgement on the effectiveness of the calculus developed thus far.

It is useful to begin by listing the elementary natural isomorphisms. For this purpose we use a home-grown, but hopefully self-evident, lambda notation.

- (88) $\lambda(R :: \perp\!\!\!\perp) \cong \lambda(R :: R \times \perp\!\!\!\perp)$
 (89) $\lambda(R :: R + \perp\!\!\!\perp) \cong \lambda(R :: R)$
 (90) $\lambda(R, S :: R + S) \cong \lambda(R, S :: S + R)$
 (91) $\lambda(R, S, T :: R + (S + T)) \cong \lambda(R, S, T :: (R + S) + T)$
 (92) $\lambda(R, S :: R \times S) \cong \lambda(R, S :: S \times R)$
 (93) $\lambda(R, S, T :: R \times (S \times T)) \cong \lambda(R, S, T :: (R \times S) \times T)$
 (94) $\lambda(R :: R) \cong \lambda(R :: R \times \mathbf{1})$
 (95) $\lambda(R, S, T :: R \times (S + T)) \cong \lambda(R, S, T :: (R \times S) + (R \times T))$

Of these (88) is trivial (the isomorphism is $\perp\!\!\!\perp$ itself) and (89) and (95) we have already discussed. Hints on how to prove (90)-(94) are given below. Of course the reader may wish to ignore the hints altogether.

Hints: Isomorphisms (90) and (91) can be constructed using the same strategy as that used to construct (95). In the case of (90) the very short calculations that are necessary can be made yet shorter by noting that the constructed isomorphism is its own reverse.

Isomorphisms (92) and (93) require a somewhat different strategy. The reason is that the natural isomorphism properties of split and product only help in the construction of up-simulations

(elements of $R \triangleleft S$ for given R and S) and not natural transformations. Moreover, whereas the calculation of the right domain of a split is straightforward, the calculation of its left domain is not (compare 61(a) with 61(b)). To add to this, split preserves imps but does not preserve co-imps. One may avoid all these difficulties by constructing two up-simulations, one of the left-side relator by the right-side relator and one of the right-side relator by the left-side relator. (In the case of (92) these "two" up-simulations obviously coincide.) For this purpose the naturality properties are used. One then proves that the first is the reverse of the second. It then suffices to prove that both are imps and to calculate the right domains of each.

The proof of (94) is a case apart. Note that \ll is an up-simulation of the identity relator by $\times 1$. Try restricting \ll so that its right domain is $I \times 1$ and then verify that your conjectured isomorphism meets all the requirements. *End of hints*

Having completed this task you should be able to verify the following properties of the constructed isomorphisms. (The names $\alpha_1 \dots \alpha_8$ have been given to the isomorphisms in order of their appearance in the list above.)

- (96) $\quad \quad \quad \perp\!\!\!\perp \quad = \quad \alpha_1 \circ R \triangle \perp\!\!\!\perp$
- (97) $\quad \quad R \triangleright \perp\!\!\!\perp \circ \alpha_2 \quad = \quad R$
- (98) $\quad \quad R \triangleright S \circ \alpha_3 \quad = \quad S \triangleright R$
- (99) $\quad R \triangleright (S \triangleright T) \circ \alpha_4 \quad = \quad (R \triangleright S) \triangleright T$
- (100) $\quad \quad \quad R \triangle S \quad = \quad \alpha_5 \circ S \triangle R$
- (101) $\quad \quad R \triangle (S \triangle T) \quad = \quad \alpha_6 \circ (R \triangle S) \triangle T$
- (102) $\quad \quad \quad R \quad = \quad \alpha_7 \circ R \triangle !$

Rest assured: all have very trivial proofs. Note the pattern: the relators $+$ and \times have been systematically replaced by their corresponding morphism and 1 has been replaced by its morphism $!$. The α 's are eaten up on the right side by a junc and on the left side by a split.

Consider now the quaternary relators F and G , respectively, defined by

$$R, S, T, U \mapsto (R+S) \times (T+U)$$

$$R, S, T, U \mapsto (R \times T) + (S \times U)$$

We conclude this section by showing that F simulates G .

We begin by constructing γ satisfying requirement (a) of a natural simulation (see definition 84).

$$\begin{aligned}
 & \gamma \in (R+S) \times (T+U) \triangleleft (R \times T) + (S \times U) \\
 \Leftarrow & \quad \{ \gamma := \beta_1 \triangleright \beta_2, \text{ naturality of junc } \} \\
 & \beta_1 \in (R+S) \times (T+U) \triangleleft R \times T \\
 \wedge & \quad \beta_2 \in (R+S) \times (T+U) \triangleleft S \times U \\
 \Leftarrow & \quad \{ \beta_1 := \alpha_1 \times \alpha_2, \beta_2 := \alpha_3 \times \alpha_4, \text{ naturality of product } \} \\
 & \alpha_1 \in R+S \triangleleft R \wedge \alpha_2 \in T+U \triangleleft T \\
 \wedge & \quad \alpha_3 \in R+S \triangleleft S \wedge \alpha_4 \in T+U \triangleleft U \\
 \Leftarrow & \quad \{ \text{naturality of the injections } \} \\
 & \alpha_1 = \alpha_2 = \hookrightarrow \wedge \alpha_3 = \alpha_4 = \hookrightarrow
 \end{aligned}$$

We have thus shown that

$$(\hookrightarrow \times \hookrightarrow) \nabla (\leftarrow \times \leftarrow) \in (R+S) \times (T+U) \leftrightarrow (R \times T) + (S \times U)$$

We continue to call the constructed spec γ .

Clearly, γ is an imp; it is, however, not a co-imp and cannot therefore be an isomorphism. Calculation of its right domain using (79), (60) and (77) is straightforward. We obtain

$$\gamma^> = (I \times I) + (I \times I)$$

as required. The verification that F simulates G is thus complete.

In just the same way that we explored the behaviour of natural isomorphisms with respect to split and junc, it is useful to explore further the properties of the simulation γ . First, in a matter of a few steps using junc-sum and product-split fusion followed by the junc-split abide law and the definition of sum, one obtains

$$\gamma \circ (R \triangle T) + (S \triangle U) = (R+S) \triangle (T+U)$$

Second, using the definition of product, the junc-split abide law and the definition of sum one obtains:

$$\gamma = (\ll + \ll) \triangle (\gg + \gg)$$

which is a better form of γ for the final calculation which is to verify that

$$(R \nabla S) \times (T \nabla U) \circ \gamma = (R \times T) \nabla (S \times U)$$

(This calculation also takes only a few steps and involves using the fusion laws and the definition of product.)

This concludes our discussion of natural simulations and of the elementary properties of the polynomial relators.

References

- [1] R.C. Backhouse. On a relation on functions. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business*. Springer-Verlag, 1990.
- [2] R.C. Backhouse, Bruin P. de, P. Hoogendijk, G. Malcolm, Voermans T.S., and J. van der Woude. A relational theory of datatypes. In preparation: copies of draft available on request, 1991.
- [3] R.C. Backhouse, Bruin P. de, G. Malcolm, Voermans T.S., and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*. Elsevier Science Publishers B.V., 1991.
- [4] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and non-deterministic programs. *Theoretical Computer Science*, 43:123-147, 1986.
- [5] P.J. de Bruin. Naturalness of polymorphism. Technical Report CS8916, Department of Mathematics and Computing Science, University of Groningen, 1989.
- [6] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1986.
- [7] L. Meertens. Constructing a calculus of programs. In J.L.A. van de Snepscheut, editor, *Conference on the Mathematics of Program Construction*, pages 66-90. Springer-Verlag LNCS 375, 1989.
- [8] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513-523. Elsevier Science Publishers, 1983.
- [9] Willem Paul de Roever Jr. *Recursive program schemes: semantics and proof theory*. PhD thesis, Free University, Amsterdam, January 1974.
- [10] G. Schmidt and T. Ströhlein. Relation algebras: Concept of points and representability. *Discrete Mathematics*, 54:83-92, 1985.
- [11] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture*, ACM, London, September 1989.

On Adding Algebraic Theories with Induction to Typed Lambda Calculi (Extended Abstract)

Val Breazu-Tannen

Ramesh Subrahmanyam ¹

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA
E-mail: val@cis.upenn.edu, ramesh@saul.cis.upenn.edu

1 Introduction

In the design and analysis of a functional programming language, the study of the logic of equality of expressions in the language is an important aspect. Many functional programming languages such as ML also provide an abstract datatype facility. At the level of program specification and reasoning, the interface between the implementation and the use of data abstraction consists of a signature of datatype operations and a system of axioms about data expressions. The logic of such interfaces is usually taken to be an equational algebraic logic [Ehrig & Mahr 1985, Goguen & Meseguer 1985]. In order to reason about the abstract datatypes *and* the functional language that is hosting them, we mix the equational theories of the typed lambda calculi with the algebraic equational theories of the datatypes. It is natural and significant to ask whether the resulting mixed (or, sum) equational theory is a conservative extension of the component calculi. If such a result is obtained one could say that a conceptual separation between the two logics has been obtained.

If we consider the interaction of the simply typed lambda calculus and algebraic equational theories, such a result was obtained in earlier work [Breazu-Tannen & Meyer 1987]. Furthermore in [Breazu-Tannen 1988], we reduced equality of *mixed terms* (which are simply typed terms over the set of algebraic operations viewed as higher-order constants) via a Turing reduction, to the equality of algebraic terms in the algebraic equational theory. We call such a result a *separation theorem*. The conservativity of the mixed theory over the two components follows as two corollaries. We do not have yet an analogous result in the case of the interaction of the simply typed lambda calculus with the equational theory of a given set of algebraic equations with the rule of structural induction. However, we do have conservative extension results and a proof-theoretic analysis which ultimately we believe will lead to such a result, as was the case with the result in [Breazu-Tannen 1988] which was preceded and inspired by the proofs of conservative extension [Breazu-Tannen & Meyer 1987]. From a technical point of view, we are introducing new ideas for applying the proof strategy of [Breazu-Tannen & Meyer 1987] for conservative extension problems relating to the simply typed lambda calculus. Moreover, we do have a normal form theorem proved using essentially the same techniques that were developed for the conservative extension proof.

¹The authors are partially supported by ONR Grant N00014-88-K-0634, by NSF Grant CCR-90-57570, and by an IBM Graduate Fellowship.

2 Is λ^- Conservatively Extended?

Though we do not have a full answer to this question, there is a simple semantic proof that, given an algebraic theory with an infinite initial algebra and the property that every ground expression is provably equal to some constructor expression (call this Property C), λ^- is indeed conservatively extended. To see this, observe the well-known fact that the rule of induction is sound with respect to the initial algebra \mathcal{I} (assuming Property C). Next observe that $\beta\eta$ and equational reasoning is sound for reasoning about equality of λ -terms in the full type hierarchy over \mathcal{I} , and the soundness of structural induction extends to this as well. If e is an equation between two λ -terms, provable by equational reasoning, structural induction, the algebraic axioms and $\beta\eta$ (i.e. the composite system), then it is true in the full type hierarchy. Friedman's completeness theorem (see [Friedman 1975]) solves the problem: an equation between two pure λ -terms is true in the full type hierarchy over an infinite base type if and only if it is provable by $\beta\eta$ -equational reasoning. It follows that

Proposition 1 *If T is an algebraic theory with an infinite initial algebra, and satisfying Property C, then λ^- is conservatively extended by T plus the rule of structural (algebraic) induction.*

3 Conservative Extension by λ^-

The general technique for obtaining conservative extension of the algebraic theory (with induction) by λ^- is as follows. Given a higher-order proof of an algebraic equation, we set up a sequence of transformations that convert it to an inductive algebraic proof. For a detailed description of these transformations, see the full paper ([Breazu-Tannen & Subrahmanyam 1989]).

Though the equation proved is of base type, there may be replacement steps in the proof the types of whose premises and conclusions may be different. The presence of such replacement rule instances allows for the possibility of higher type free variables in the proof. The first transformation CBT converts the given proof into one in which every term is of base type.

A replacement rule may cause the free variables in the premises to be captured by some lambda-abstraction in the context of replacement. The premises of a transitivity rule instance may have free variables that may not occur in the conclusion; similarly, for the substitutivity rule. Thus while the equation proved itself may have no higher-order free variables, it is possible for higher-order free variables to appear in other terms in the proof. The transformation *Rem-Free-Vars* transforms the given proof into one in which there are no higher-order free variables, that "disappear" at transitivity steps. The transformation *Push-Subst* gets rid of free variables that appear in the premise of some substitutivity step, but not in its conclusion.

Transformations for removing occurrences of free variables that get bound in some context at a replacement step are complex. The most obvious such transformation, given a node in a proof where a replacement in such a context takes place, is one which attempts to insert every term in the proof above the said node, in the said context. This runs afoul of the induction steps, since such naive insertions may capture the induction variable at the step.

It is not hard to see that any term of base type in β normal form containing no higher-order free variables is in fact an algebraic term. The point of such transformations therefore is to obtain proofs in which no term has higher-order free variables, and in which every term is of base type and in β -normal form.

Once we have removed some of the free-variables via **Rem-Free-Vars** and **Push-Subst**, we must be careful that any later transformations do not reintroduce the problems that these transformations solved. The two operations of removing free variables of higher-type that occur in the premise of some replacement rule, but not in its conclusion, and of transforming the proof into one in which every term is in normal form must ensure that. In this paper, these two operations are achieved simultaneously by iterating a certain sequence of transformations. Most importantly, the iteration should terminate.

An important observation is this: if we can obtain a proof in which every term is of base type, is in normal form and there are no higher-order free variables that occur in the premises of a transitivity or substitutivity rule but not in the conclusion, then there are no higher-order free variables that occur in the premise of a replacement rule but not in its conclusion. Thus it suffices to focus our attention on obtaining a proof in which every term is in normal form. This is the reason for coupling the two operations, as mentioned in the previous paragraph.

First, let us consider a higher-order proof of an algebraic equation which has no replacement rule instances with premises of higher types. If one normalizes every term in such a proof, one obtains a tree which is a proof in a system obtained by changing the usual replacement rule to a *generalized replacement rule* (the context $C[]$ may have more than one hole):

$$\frac{t = t'}{C[t, \dots, t] = C[t', \dots, t']} \quad \text{repl}$$

However, if we have replacement rules with premises that are of higher type, the resultant tree is no longer a proof tree. The remaining problem is that, what were previously replacement steps are no longer replacement rule instances:

$$\frac{\text{nf}(t_1) = \text{nf}(t_2)}{\text{nf}(C[t_1]) = \text{nf}(C[t_2])}$$

We must replace this by a deduction which does not reintroduce the problems that were gotten rid of by earlier transformations. Towards this end we define the following concept:

Definition 2 (Synchronous Reductions) Let $C[]$ be a context, and t_1 and t_2 be two terms that have the same type as the hole in $C[]$. We say that a pair of reductions (p, q) on the pair of terms $(C[t_1], C[t_2])$ is synchronous if the redexes of p and q occur at the same address in $C[t_1]$ and $C[t_2]$, which must be an address internal to the context $C[]$.

Now we can prove the following lemma:

Lemma 3 (Filler Lemma) Let $C[]$ be a given context with a single hole. Let t_1 and t_2 be two normal form terms such that

$$\begin{aligned} C[t_1] &\xrightarrow{p_1} D_1 \\ C[t_2] &\xrightarrow{p_2} D_2 \end{aligned}$$

where (p_1, p_2) are synchronous reductions. Then there is a deduction of $D_1 = D_2$ from $t_1 = t_2$.

We can now use the filler lemma as follows: Consider a replacement step, after we have normalized its terms. Let $C[t_1] \xrightarrow{p_1} D_1$ and $C[t_2] \xrightarrow{p_2} D_2$ constitute a synchronous reduction. Let D_1^i ($0 \leq i \leq m_1$) and D_2^i ($0 \leq i \leq m_2$) constitute two β -reduction sequences with $D_j^i \rightarrow_\beta D_j^{i+1}$, $D_j^0 = D_j$ ($j = 1, 2$). Observe that $D_j^{m_j} = \text{nf}(C[t_j])$ ($j = 1, 2$). We can construct the following deduction:

$$\frac{\frac{\text{nf}(t_1) = \text{nf}(t_2)}{\dots D_1^i = D_1^{i-1} \dots D_1 = D_2 \dots D_2^i = D_2^{i+1} \dots}}{\text{nf}(C[t_1]) = \text{nf}(C[t_2])}$$

Now, we have to repeatedly *insert* such deductions at the problematic replacement steps mentioned before. The resultant tree is a proof tree. Not all its terms are in normal form, and the tree now has some substitution steps with higher-order free variables occurring in the premises but not in the conclusion. We will thus have to iterate the process of applying Push-Subst and the above insertion. Will this iteration terminate? Yes, but only for a particular sequence in which synchronous deductions are computed while doing the above transformation. Very importantly, note that, using the length of longest reduction sequence from a term as a measure, after insertion, the terms in the conclusion of every replacement step have lesser measure than the terms in the conclusion step prior to the insertion. However, a rigorous argument for termination of the transformation is a little more complicated, and we refer the reader to the full paper [Breazu-Tannen & Subrahmanyam 1989].

As noted before, we eventually have a proof tree in which every term is of base type and in normal form, and in which there are no higher-order free variables that appear in the premises of a replacement step but not in its conclusion. Such terms must all be algebraic, and we have, thus, shown that any (possibly higher-order) proof of an algebraic equation using induction, can be transformed to an algebraic proof using induction. This yields the following theorem:

Theorem 4 *Any given algebraic theory with structural induction is conservatively extended by the theory of the simply typed lambda calculus λ^{\rightarrow} .*

The transformations indicated also yield the following normal-form theorem :

Theorem 5 *Fix an algebraic theory E , a structural induction rule IND and two terms t_1 and t_2 . $\lambda^{\rightarrow} E + IND \vdash t_1 = t_2$ if and only if there exists in $\lambda^{\rightarrow} E + IND$ a proof of $\beta \text{nf}(t_1 \vec{x}) = \beta \text{nf}(t_2 \vec{x})$, such that \vec{x} are fresh variables. $t_1 \vec{x}$, $t_2 \vec{x}$ are of base type, and such that every term in the proof is in β normal form.*

4 Summary and Future Work

In summary, we establish the following results:

1. Conservative extension of algebraic theories with the rule of structural induction by the simply typed lambda calculus, λ^{\rightarrow} .
2. Conservative extension of λ^{\rightarrow} by algebraic theories and structural induction, provided the algebraic theory has an infinite initial algebra, and the set of constructors is so chosen that every ground term is equal to some constructor term.

3. A normal form theorem for proofs in the mixed theory.

In a sequel to this paper we plan to show the conservative extension of algebraic theories and structural (algebraic) induction by the polymorphic lambda calculus, λ^V . The problems that we faced in extending the proof in [Breazu-Tannen & Meyer 1987] in the presence of induction arise in attempting a conservative extension for the polymorphic lambda calculi, and we believe that essentially the ideas of synchronous reduction couple with the idea of quasi-simple terms introduced in previous work, can be used to prove the same.

References

- [Breazu-Tannen & Meyer 1987] V. Breazu-Tannen and A. R. Meyer. Computable values can be classical. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, pages 238–245, ACM, January 1987.
- [Breazu-Tannen & Subrahmanyam 1989] V. Breazu-Tannen and R. Subrahmanyam. *Lambda Calculus, Structural Induction, and Conservative Extension*. Tech. Rep. MS-CIS-89-64/LOGIC & COMPUTATION 13, Department of Computer and Information Science, University of Pennsylvania, 1989.
- [Breazu-Tannen 1988] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the Symposium on Logic in Computer Science*, pages 82–90, IEEE, July 1988.
- [Ehrig & Mahr 1985] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1: equations and initial semantics*. Springer-Verlag, 1985.
- [Friedman 1975] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloquium '73*, pages 22–37, *Lecture Notes in Mathematics*, Vol. 453, Springer-Verlag, 1975.
- [Goguen & Meseguer 1985] J. Goguen and J. Meseguer. Initiality, induction, and computability. In *Algebraic Methods in Semantics*, Cambridge Univ. Press, 1985.

Extracting Recursive Programs in Type Theory

Scott F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

Constructive Type Theory (CTT) [CAB⁺86, Mar84] is a foundational theory of mathematics and programming, originated by Per Martin-Löf. The key to using type theory as a logic is the *propositions-as-types* principle, whereby propositional assertions are directly expressed by types of type theory; a proposition is true just when its type interpretation is inhabited, i.e. is non-empty.

Type theory is of use to computer scientists when we place the proof as central rather than the theorem: the proof is partly a program, the object in the type, and in the process of proving a theorem a program can be automatically *extracted* (see [CAB⁺86]), and furthermore this program is guaranteed correct.

One weakness of the use of the extraction paradigm to date, however, is that it is impossible to extract arbitrary recursive programs from proofs, because all objects in types must be total, whereas many general recursive functions are not total. We give here an extension to type theory, adding new types and principles thereupon, so that extraction of recursive programs is possible.

Types for computations which may have no value are added. This is accomplished by adding a new type constructor: given a type A , the type \overline{A} ("A bar") is the type of computations over A . A naive interpretation of \overline{A} would be $A \cup \{\perp\}$. Traditional constructive type theories only have type for total computations, and type systems for programming languages naturally only type partial computations; what we achieve here is the union of these two notions of type, bridged by the constructor $\overline{}$. Preliminary results appear in [CS87].

The fixed point typing principle allows recursive functions to be typed.

PRINCIPLE 1 (FIXED POINT TYPING) *Letting $A \xrightarrow{P} B$ be $\overline{A \rightarrow B}$, if a functional f is in the type $(A \xrightarrow{P} B) \rightarrow (A \xrightarrow{P} B)$, then $Y(f)$, the fixed point of f ,*

is in the type $A \xrightarrow{P} B$.

This principle extends the extraction paradigm to allow recursive programs to be extracted from proofs. This in turn makes the extraction paradigm, one of the unique features of type-theoretic reasoning, potentially more useful as a component of a system for developing correct programs.

We sketch how these new concepts may be used for reasoning, with particular emphasis on proving by extracting fixed point objects. Propositions below are expressed as types, as is standard in type theory; for instance, $\forall x:A. B \stackrel{\text{def}}{=} x:A \rightarrow B$, a dependent function space.

The partial type operator \bar{A} gives a general notion of partiality. For example, consider the (total) function space on natural numbers $N \rightarrow N$; there are seven partial type versions, $\bar{N} \rightarrow \bar{N}$, $N \rightarrow \bar{N}$, $\bar{N} \rightarrow N$, $\bar{N} \rightarrow \bar{N}$, $\bar{N} \rightarrow N$, $\bar{N} \rightarrow \bar{N}$, and $\bar{N} \rightarrow \bar{N}$. $N \rightarrow \bar{N}$ is the type we will use to represent partial functions, notated $x:A \xrightarrow{P} B(x) \stackrel{\text{def}}{=} x:A \rightarrow \bar{B}(x)$.

The bar operator may be applied to types which represent propositions, giving types such as $\forall n:N. \exists m:N. \bar{P}(m, n)$, which are called *partial propositions*. \bar{A} for any type A is trivially true under the propositions-as-types interpretation, because $\perp \in \bar{A}$. However, if $a \in \bar{A}$ and $a \downarrow$, then $a \in A$, so A is true. If we can potentially show termination of the extract object a , proving a partial proposition is useful.

Universally quantified partial propositions are the most relevant, because their extract objects are functions—a partial function type has as analogue a partial universal quantifier $\bar{\forall}x:A. B(x) \stackrel{\text{def}}{=} x:A \xrightarrow{P} B(x)$.

One important extension to the extraction paradigm is the extraction of arbitrary recursive computations. This is significant, because previously it was not possible to automatically extract recursive programs. For instance, take the proposition

$$\forall l:\text{List}. \exists l':\text{List}. \text{Ordered}(l') \ \& \ \text{Permutation}(l, l').$$

It is not possible to extract an order $n \cdot \log n$ sorting routine, because the only way to prove this theorem is by double induction on natural numbers which always produces an order n^2 algorithm. An order $n \cdot \log n$ program can feasibly be extracted from the corresponding partial proposition.

A sketch of a recursive primality tester being extracted from a partial proposition is now given.

DEFINITION 2 Define three predicates,

$$\begin{aligned}
y \text{ Divides } x &\stackrel{\text{def}}{=} \exists z:N. y * z = x \\
x \text{ Is_Prime_Thru } y &\stackrel{\text{def}}{=} \forall z:N. 1 < z \leq y \Rightarrow \neg(z \text{ Divides } x) \\
x \text{ Is_Composite_Thru } y &\stackrel{\text{def}}{=} \exists z:N. 1 < z \leq y \ \& \ z \text{ Divides } x
\end{aligned}$$

LEMMA 3 *We may show*

$$\vdash \forall x:N. \forall y:N. x \text{ Is_Prime_Thru } y \vee x \text{ Is_Composite_Thru } y, \text{ extract } e.$$

The extract object e will be a recursive function such that $e(x)(x-1)$ decides whether or not x is prime; furthermore, if x is composite, one of its factors will be returned.

PROOF. Using the fixed point typing principle, it suffices to show

$$\frac{z:(\forall x:N. \forall y:N. x \text{ Is_Prime_Thru } y \vee x \text{ Is_Composite_Thru } y), x:N, y:N \vdash x \text{ Is_Prime_Thru } y \vee x \text{ Is_Composite_Thru } y}{\text{extract } e'},$$

where $e \stackrel{\text{def}}{=} Y(\lambda z.e')$. This means we are extracting a recursive function from this proof; inside e' , applications of z are recursive calls, which proof-theoretically amount to uses of the hypothesis z . Proceed by cases on $y \text{ Divides } x$ (this predicate is decidable).

CASE $y \text{ Divides } x$: Clearly then x is composite, so the right disjunct can be proven.

CASE $\neg(y \text{ Divides } x)$: Recursively use the hypothesis z for y one smaller, i.e. apply $z(x)(y-1)$, and the result also follows.

QED.

Partial propositions extend the collection of statements that can be phrased when reasoning constructively, giving a more expressive logic. Since there is no construction to validate statements in classical logic, the notion of partial proposition makes no sense there. Automatic extraction of recursive programs from partial propositions is a potentially useful methodology for deriving correct programs.

Partial types and partial propositions also significantly narrow the gap between programming and proving, and we believe exciting possibilities for programming systems arise from this. Many of the properties of proof construction systems for type theory such as Nuprl [CAB⁺86] give rise to interesting program construction systems: programs are typed before they are actually written (the opposite of the type synthesis of ML), and programs (and subprograms) are automatically annotated with the properties they must satisfy.

References

- [CAB⁺86] R. L. Constable, S. Allen, H. Bromley, W. R. Cleveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. P. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [CS87] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.
- [Mar84] Martin-Löf, Per. *Intuitionistic Type Theory*. Bibliopolis, 1984.

Using Algebraic Specifications in Floyd-Hoare Assertions*

Hantao Zhang, Angshuman Guha, and Xin Hua

Department of Computer Science

The University of Iowa

Iowa City, IA 52242

{hzhang,guha,xin}@herky.cs.uiowa.edu

Extended Abstract

Theoretically, the problem of program verification was solved two decades ago, because the axiomatic approach led by Floyd and Hoare reduces the problem of program verification to that of theorem proving [6, 9], and the resolution principle of Robinson is capable of proving any valid theorems [16]. In reality, the hopes to have full program verification for most sensitive areas still have not been fulfilled, even though the reasoning power of resolution-based theorem provers has been increased hundreds times compared to two decades ago.

Siekman pointed out in [18] that “the main problem of program verification is no longer the lack of strength of the deduction systems but rather the inadequacy of unstructured specification techniques.” While we share his view, we also think that the lack of the capacity of performing inductive reasoning in the deduction systems severely restricts the power of these systems in program verification. The importance of recursive or inductive techniques to programming has been pointed out by many computer scientists, including McCarthy [13], Boyer and Moore [1]. It is easy to see that many algorithms or data structures are specified recursively or inductively. However, current work based on the Floyd-Hoare approach does not take advantage of this recursive specification technique.

In the passed decade, many alternative approaches (especially those based on declarative languages) fully use the recursive specification technique. One of these approaches is the well-known algebraic specification approach for abstract data types [7]. The verification of those declarative programs becomes feasible when inductive reasoning is used, [1] and [15].

In this paper, we propose a simple technique which combines the Floyd-Hoare approach with the algebraic specification approach together. In the Floyd-Hoare axiomatic approach, assertions about program states are usually specified, in a straightforward way, as formulas of first-order predicate calculus. In the algebraic specification approach, recursive equations are used to specify abstract data types. We suggest to use algebraic equations as a specification language for program states. That is, we use new predicates to describe program states — these predicates are specified by recursive equations and be used in Hoare assertions.

The advantage of this combination is at least twofold: (a) Assertions become more abstract (i.e., neater and more concise). In particular, quantifiers in the assertions can be avoided. (b) Some useful lemmas about recursively-defined predicates can be used, because their validity can be easily established by inductive reasoning.

*Partially supported by the National Science Foundation Grants no. CCR-9009414 and INT-9016100.

Traditionally, quantifiers have to be removed by skolemization when assertions are input to a resolution-based prover. Skolemization often makes formulas complicated by introducing new skolem functions and duplicating subformulas. If we use recursively defined predicates instead of quantifiers to specify assertions, then skolemization can be totally avoided and assertions will become simpler for a theorem prover to prove. In fact, avoiding quantifiers is the only heuristic we used to introduce new predicates.

In the paper, we illustrate the above idea by three examples of program verification:

1. **Initializing an array to zeros:** This program was used in a recent paper by Chisholm, Smith and Wojcik, for illustrating typical problems of proving program assertions using a resolution-based theorem prover [2]. The C program given in [2] is quite trivial, though the automatic proof of the related assertions (a loop-invariant and a post condition) is quite complicated: it needs special strategies and intensive user's assistance in the automated reasoning system ITP, a well-known resolution-based theorem prover built at Argonne National Laboratory [12]. We show that the verification of the same program becomes trivial when we choose right formulation of assertions.
2. **Sorting an array by insertion:** This program is taken from Dromey's latest book [5]. The book presents the state-of-art of program synthesis, i.e., how to derive programs from the pre- and post-conditions through a sequence of refinements. This process of derivation attempts at specifying the loop invariants, too, when loops are constructed. We believe that this idea holds merit as it seems to answer an old 'cry': "the disadvantage of Hoare's proof rules is that it forces you to find an invariant of each loop" [8]. But even an experienced programmer may generate loop invariants in the derivation process that are not always strong enough for a proof of correctness. We found such problems in the book as well as in our experience of program derivation. A proof of assertions (either by hand or by machine) can reveal all the missing parts. Nevertheless, one could argue in favour of this program derivation approach that it provides with loop invariants which might not be exactly sufficient, but at least close to it: better than finding loop invariants from scratch!
3. **Sorting an array by exchange:** This program was taken from Musser's recent report [14]. He suggests that a pragmatic solution to program verification consists of three main elements:
 - Concentration on *small but very widely useful modules*.
 - Recognition that one can still gain much useful assurance about the correctness of computations from the judicious use of *simplified mathematical models* of those computations.
 - Avoiding too much reliance, in the short term, on automated verification systems; instead, using *highly detailed hand proofs* to experiment with notations, theory development, and proof presentation techniques.

A detailed hand proof of the exchange-sort program was given in [14] to illustrate the above ideas. By choosing right predicates, we are able to automatically check Musser's proof. That is, by using the recursively defined predicates in the assertions, Musser's proof becomes much simplified. We believe this kind of simplification is not unusual when equations are used as a specification language in the Floyd-Hoare approach.

We assume the reader's knowledge about Hoare's proof rules [9]. By *correctness* of programs, we mean *partial correctness*. The problem of termination of programs is not discussed in this

paper; it can be better handled by the complexity-of-algorithm approach. The correctness of the above three programs has been automatically checked in the theorem prover *RRL — Rewrite Rule Laboratory* [10, 11]. The method used in *RRL* is called *clausal superposition* [20], which converts each clause into a conditional rewrite rule (choose a maximal literal as the head of the rewrite rule and put the negation of the remaining literals into the condition). Superposition between rewrite rules is similar to either resolution [16] or paramodulation [17] between clauses. Rewriting on clauses is more powerful than subsumption and demodulation together; though each rewriting can be simulated by several steps of resolution or paramodulation. For inductive reasoning, the cover-set induction method proposed in [21] is used. The cover-set induction is a generalization of structure induction for equational systems. It employs many ideas of the Boyer-Moore prover [1].

The work presented in this paper is just one of the first attempts to combine different approaches to program verification. We believe that how to apply the verification techniques developed for declarative languages to imperative languages is a fruitful research topic. For instance, suppose the quicksort algorithm is specified using the following two equations:

$$\begin{aligned} \text{qsort}([\]) &= [\]; \\ \text{qsort}([x \mid y]) &= \text{append}(\text{qsort}(\text{splitlow}(x, y)), [x \mid \text{qsort}(\text{splithigh}(x, y))]), \end{aligned}$$

where *splitlow*(*x*, *y*) returns all the elements in *y* which are less than or equal to *x*, and *splithigh*(*x*, *y*) returns all the elements in *y* which are greater than *x*. Then it takes a theorem prover (say, *RRL*) less than a minute (on a Vax 11/780) to prove the correctness of *qsort* [21] (it involves proving theorems like *sortedp*(*qsort*(*x*)) and *permutation*(*x*, *qsort*(*x*)) by induction). On the other hand, it is very difficult to prove the correctness of the quick-sort algorithm implemented in an imperative language (a computer checked proof has been reported in [15]). If we have a mechanism to keep a one-to-one mapping between a list and an array, then we can still gain much useful assurance about the correctness of the quick-sort algorithm based on the use of arrays.

It will also be interesting to further study algebraic specifications from the viewpoint of its application in the Floyd-Hoare approach. For instance, a procedure can be generally considered as an algebraic function. However, many procedures require nontrivial pre-conditions. In this case, these procedures correspond to partial functions. It is known that the initial semantics of algebraic specifications are unsuitable for handling partial functions. To provide a good semantics to the algebraic specifications of Hoare assertions, we plan study semantic tools which can handle partial functions, such as final algebras, partial algebras, order-sorted algebras, or the constructor model interpretation [19].

References

- [1] Boyer, R.S., and Moore, J.S. (1979). *A computational logic*. Academic Press, New York.
- [2] Chisholm, G.H., Smith, B.T., and Wojcik, A.S., (1989). "An automated reasoning problem associated with proving claims about programs using Floyd-Hoare inductive assertion methods", *J. of Automated Reasoning*, 5, 533-540.
- [3] Dijkstra, E.W., (1975). "Guarded Commands, Nondeterminacy and the Formal Derivation of Programs", *Communications of the ACM*, 18, 263-267.
- [4] Dijkstra, E.W., (1976). *A Discipline of Programming*. Prentice Hall, Englewood Cliffs.

- [5] Dromey, Geoff, (1989). *Program Derivation - The Development of Programs from Specifications*. Addison Wesley.
- [6] Floyd, R., (1967). "Assigning Meaning to Programs", *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, pp 19-32.
- [7] Guttag, J.V., Horowitz, E., and Musser, D.R., (1978). "Abstract data types and software validation". *Communication of ACM*, V. 21, No. 12, 1048-1063.
- [8] Gries, David, (1981). *The Science of Programming*. Springer-Verlag New York Inc.
- [9] Hoare, C. A. R., (1969). "An Axiomatic Approach to Computer Programming", *Communications of ACM*, 12, 576-580, 583.
- [10] Kapur, D., and Zhang, H., (1989). "An overview of RRL: Rewrite Rule Laboratory", *Proc. of the third International Conference on Rewriting Techniques and its Applications* (RTA-89), April 1989, Chapel Hill, NC, Springer Verlag LNCS 355, 513-529.
- [11] Kapur, D. and Zhang, H., (1989). "RRL: A Rewrite Rule Laboratory, User's Manual", *Revised Version*, Report 89-03, Dept. of Computer Science, The University of Iowa.
- [12] Lusk, E.L. and Overbeek, R.A., (1984). "An LMA-based theorem prover". Technical Report ANL-82-84, Argonne National Laboratory, Argonne, IL.
- [13] McCarthy, J., (1963). "Basis for a mathematical theory of computation". In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds., North-Holland Pub. Co., Amsterdam. 33-70.
- [14] Musser, D.R., (1989) "Elements of a pragmatic approach to program verification", Technical Report 89-24, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY.
- [15] Pase, W., and Kromodimoeljo, S., (1987) *m-NEVER User's Manual*. TR-87-5420-13, I.P. Sharp Associates Limited, Ottawa, Canada.
- [16] Robinson, J.A., (1965) "A machine-oriented logic based on the resolution principle", *J. of ACM*, 12.
- [17] Robinson, G., and Wos, L. (1969) "Paramodulation and theorem proving in first-order theories with equality", in B. Meltzer and D. Michie (eds.) *Machine Intelligence 4*, Edinburgh University Press, Edinburgh.
- [18] Siekmann, J.H., (1989) "The history of deduction systems and some applications", in K. H. Blasius and H.J. Burckert (eds.) *Deduction Systems in Artificial Intelligence*. Ellis Horwood Series in AI, Chichester.
- [19] Zhang, H., (1989) "Constructor model as abstract data types", *Proc. of First Intl. Conf. on Algebraic Methodology and Software Technology*, Iowa City, Iowa, May 1989.
- [20] Zhang, H., and Kapur, D., (1988) "First-order logic theorem proving using conditional rewrite rules", *Proc. of the 9th Intl. Conf. on Automated Deduction* (CADE-9), Argonne, Illinois. Springer Verlag LNCS 310 p. 1-20.
- [21] Zhang, H. and Kapur, D., and Krishnamoorthy, M.S., (1988) "A Mechanizable Induction Principle for Equational Specifications," *Proc. of Ninth International Conference on Automated Deduction* (CADE-9), Argonne, IL, May 1988. Springer-Verlag LNCS 310, pp. 250-265.

On the ground convergence of conditional theories

Emmanuel Kounalis and Michaël Rusinowitch
CRIN, BP239, 54506 Vandœuvre-les-Nancy, France &
INRIA Lorraine, BP 101, 54600 Villers-les-Nancy, France
e-mail: {kounalis, rusi}@loria.crin.fr

April 26, 1991

Abstract

We present a new technique for proving the ground convergence property of conditional theories. We also propose methods for *completing* a non-ground convergent theory to obtain an equivalent ground convergent one. We finally give sufficient conditions for a given equational theory to ensure that convergence on ground terms is equivalent to convergence on general terms.

1 Introduction

The ability to reason with *conditional equations* is fundamental in Computer Science. Conditional equations may represent abstract interpreters for programming languages and model formal manipulation of systems used in various applications including program verification and specifications and mechanical theorem-proving.

Let $T(F, X)$ be the set of terms built out of function symbols taken from a signature F and variables from a set X . An *equational theory* E over $T(F, X)$ is set of equations, each of which is a pair of terms in $T(F, X)$ written as $s = t$. A *conditional theory* is a set of conditional equations, each of which is either an equation or an expression C of the form $e_1 \wedge \dots \wedge e_n \Rightarrow e$, where e, e_1, \dots, e_n are equations. The equations e_1, \dots, e_n are said to be *conditions* of C and e is said to be the *conclusion* of C . Reasoning about equations may involve deciding if an equation is a consequence of a given set of conditional equations or if an equation is true in a given model. One method for solving these problems consists in compiling the (conditional) theories into a set of rewrite rules: a rewrite system R over a set of terms $T(F, X)$ is a set of directed conditional equations, called rewrite rules, each of the form $s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow l \rightarrow r$ where s_i, t_i, l, r are terms over $T(F, X)$. Rewrite rules are used to replace repeatedly subterms of a given term with equal terms until the simplest term possible is obtained. This simplest term is what we call a *normal form*. If no infinite sequences of rewrites is possible, a rewrite system is said to have the *termination* property. *Confluence* of a rewrite system is a property that ensures that no term has more than one normal form. A *convergent* rewrite system is one with both the confluence and termination properties. For a convergent rewrite system R an equation $s = t$ holds in the theory defined by R if and only if the normal forms of s and t are identical.

The Knuth-Bendix completion method [6] was designed to generate convergent rewrite systems for a given set of unconditional equations. The basic idea of this method consists of that confluence of finite terminating systems can be effectively tested by checking equality of normal forms of a finite set of pairs of terms, called *critical pairs*, formed by overlapping left-hand sides of the rules. Extension of Knuth-Bendix method to conditional theories is difficult. Some advances have been reported by [4, 12, 8, 9].

All these approaches have mainly attempted to design completion procedures for generating finite convergent systems, to determine whether two *general* terms (i.e., terms which may contain variables) rewrite to identical terms. However, for many applications, as automated theorem proving,

word problems in a given algebra, and abstract data types, the convergence property of (conditional) theories is too strong. It often suffices to have the *ground convergence* property, which means that every *ground* (i.e. variable-free) term has a unique normal form. Unfortunately ground convergence is undecidable [5] even for equational theories with only unary function symbols. The role of the ground-convergence property is deeply discussed in [10, 5].

In this paper, we **first** present a new technique for proving the ground convergence of terminating rewrite systems. The method relies on the concept of *subconvergence* of critical pairs (see definitions 4.1, 4.2, 4.3). Roughly speaking, a conditional equation is subconvergent whenever the left-hand side and the right-hand side of some of its instances reduce to the same normal form using the current rewrite system plus the equation itself, provided that the application of the latter satisfies some well-defined conditions. The instances of the critical pair to be tested are obtained through the computation of a test-set which is, in essence, a finite description of the least Herbrand model of the theory. In many cases, the subconvergence method can show the ground convergence of conditional systems. This leads to the design of general procedures to show the ground convergence property or to complete non ground convergent systems. Consider for instance, the system : $\{f(g(f(x))) \rightarrow g(f(g(x))), f(c) \rightarrow c, g(c) \rightarrow c\}$. It has been shown that there is no canonical system for this theory [5]. However it is ground convergent because the only ground constant is c and both $f(c), g(c)$ rewrite to c . The subconvergence approach try to capture this intuition as to why the system is ground convergent: computation of the test-set for the above system yields only one element, c , and this immediately ensures that the system is ground convergent. Further, if they were additional constants besides c the system would not be ground convergent anymore. This leads to our **second** result which provides sufficient conditions for the ground convergence to be equivalent to general convergence in equational theories.

The structure of this paper is as follows: Section 2 presents an overview of our approach on a simple conditional theory. Section 3 summarizes the additional basic material which is relevant to this work. For simplicity of presentation we restrict ourselves to *simplifying conditional systems* in the sense of Kaplan [4]. However most results are valid in the more general setting of *decreasing systems* in the sense of Dershowitz and Okada [2]. Section 4 introduces the concept of subconvergence and proposes tests for checking ground convergence. Section 5 provides a decidable sufficient condition for equivalence of the ground convergence and the general convergence properties of a set of unconditional equations. Section 6 discusses extensions of our method.

2 Overview of our approach: a simple example

Consider the following set of conditional equations compiled into a set R of rewrite rules:

Example 2.1

$$\begin{array}{llll} (1) & b(x) = tt & \Rightarrow & a(x) \rightarrow tt \\ (2) & q(x) = tt & \Rightarrow & a(x) \rightarrow p(x) \\ (3) & & & p(0) \rightarrow tt \end{array} \quad \begin{array}{ll} (4) & p(s(x)) \rightarrow p(x) \\ (5) & b(0) \rightarrow tt \\ (6) & q(0) \rightarrow tt \end{array}$$

Let us check whether this system is ground convergent. The key idea consists of computing critical pairs and eliminating those whose ground instances have rewrite proofs by using rules of the current system. To do it, the *first step* is to compute a test-set (see definition 3.3) which is, in our case, the set $\{0, q(s(x)), b(s(x)), s(x), tt\}$. The *next step* consists of computing the critical pairs of the system and then check them for simple subconvergence (see definition 4.1). The only one here is $c : b(x) = tt \wedge q(x) = tt \Rightarrow p(x) = tt$. We then instantiate this equation by elements of the test-set to get the conditional equations:

$$\begin{array}{ll} (6) & b(0) = tt \wedge q(0) = tt \quad \Rightarrow \quad p(0) \rightarrow tt \\ (7) & b(s(x)) = tt \wedge q(s(x)) = tt \quad \Rightarrow \quad p(s(x)) \rightarrow tt \end{array}$$

The first one is eliminated since the left-hand side $p(0)$ rewrites to tt by the rule 3. The left-hand side of the second one rewrites to $p(x)$ by applying 4 and then $p(x) \rightarrow tt$ by applying the conclusion

of the critical pair. This means that c is simply subconvergent and by theorem 4.1 the system R is ground convergent. Note that the linear completion procedure of [3] cannot be generalized to the conditional case. The critical pair generates by superposition with the rule 4 an infinite set of rules $\{b(s^n x) = tt \wedge q(s^n x) = tt \Rightarrow p(x) = tt\}$.

3 Preliminaries

3.1 Term Rewriting

Given a binary relation \rightarrow , \rightarrow^* denotes its reflexive-transitive closure, \leftarrow its inverse and \leftrightarrow its symmetric closure and \longleftrightarrow^* is its reflexive-symmetric-transitive closure. Given two binary relations, R, S , RoS denotes their composition. A *normal form* s of t for \rightarrow is denoted by $s \rightarrow^* t$. We say that two terms s and t are *joinable*, denoted by $s \downarrow t$ if there exist v such that $s \rightarrow^* v \leftarrow^* t$. The rewrite relation \rightarrow is *confluent* if $s \leftarrow^* u \rightarrow^* t$ implies that s and t are joinable. When \rightarrow is a confluent relation on the set of ground terms we say that \rightarrow is *ground confluent*. A term is *ground reducible* if it is irreducible but all its ground instances are reducible. A relation is *convergent* (resp. *ground convergent*) if it is terminating and confluent (resp. ground confluent).

In the following, we suppose that we are given a *simplification ordering* $>$ in the sense of Dershowitz [1]. An equation $s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow l = r$ is *reductive* with respect to $>$ if for all substitutions σ , $l\sigma > r\sigma, s_1\sigma, t_1\sigma, \dots, s_n\sigma, t_n\sigma$.

Given a conditional equation $s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow l = r$ there exist several ways to interpret it as a rewrite rule depending on what criterion is used to check the conditions. In this paper we use the formulation of *join systems*. Formally, to a set of reductive conditional equations R can be associated a rewrite relation \xrightarrow{R} which is recursively defined as follows [4, 2]:

Definition 3.1 Let R be a set of reductive conditional equations and let A be a term. Then $A[l\sigma] \xrightarrow{R} A[r\sigma]$ if there is a substitution σ and there is a conditional equation $a_1 = b_1 \wedge \dots \wedge a_n = b_n \Rightarrow l = r$ in R such that $\forall i \in \{1, \dots, n\} \ a_i\sigma \downarrow_R b_i\sigma$.

Whenever a reductive conditional equation $s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow l = r$ is considered as a rewrite rule it is denoted by $s_1 = t_1 \wedge \dots \wedge s_n = t_n \Rightarrow l \rightarrow r$ or $s_1 \downarrow t_1 \wedge \dots \wedge s_n \downarrow t_n \Rightarrow l \rightarrow r$. The term l (resp. r) is then called the left-hand side (resp. right-hand side) of the rule. The reductivity hypothesis ensures that any rewrite relation \xrightarrow{R} is decidable and terminating.

Given two rules $c \Rightarrow l \rightarrow r$ and $d \Rightarrow m \rightarrow s$ in a conditional system R such that l unifies via most general unifier μ with a non-variable subterm of m , then the conditional equation $c\mu \wedge d\mu \Rightarrow m\mu[r\mu] = s\mu$ is a *critical pair* of R .

We also introduce a new notion of rewriting. For checking the conditions of a conditional rules to be applied we extend the current theory R by another one E which usually contains some induction hypotheses. Hence proving a goal with this new rewriting relation and with reductive rules allows to use implicitly smaller instances of the goal itself.

Definition 3.2 Let R and E be reductive sets of conditional equations. Let A be a term, and n an occurrence of A . We write $A[l\sigma] \xrightarrow{R[E]} A[r\sigma]$ if σ is a substitution and there is a conditional equation $a_1 = b_1 \wedge \dots \wedge a_n = b_n \Rightarrow l = r$ in R , such that $\forall i \in \{1, \dots, n\} \ a_i\sigma \downarrow_{R \cup E} b_i\sigma$.

3.2 Test sets

The height of a term x will be denoted by $|x|$.

Definition 3.3 Given a set of conditional rules R , a *test-set* $S(R)$ for R is a finite set of R -irreducible terms such that:

- 1 for any R -irreducible ground term s , there exist a term t in $S(R)$ and a substitution σ such that $t\sigma = s$.

- 2 every non-ground (sub)term in $S(R)$ has infinitely many R -irreducible ground instances.
- 3 any non-ground term t in $S(R)$, has variables only at height greater or equal than $|R| - 1$ if R is left-linear and at height greater or equal than $|R|$ if R is not left-linear.

Definition 3.4 A test-substitution w.r.t. $S(H)$ is a substitution which substitutes every variable by an element of the test-set $S(H)$.

The construction of test-sets for equational theories is decidable and may be performed in a relatively efficient way. The algorithm is based on pumping lemmas in tree languages and is detailed in [7]. Such an algorithm does not exist for conditional theories. However, in we have given a method to derive test-sets in conditional theories defined over a free set of constructors.

4 The subconvergence method

Let us now introduce a powerful method for testing ground convergence of a given conditional theory. The idea consists of detecting the critical pairs that do not change the joinability relation on ground terms. For this purpose we define three criteria which are all based on the simplification of instances of critical pairs by test-substitutions. These criteria differ in the rules that are used for simplification: the *simple subconvergence* criterion make use of the conclusions of critical pairs to simplify, the *subconvergence* criterion uses the critical pairs themselves to simplify and the *relative subconvergence* criterion uses the premisses of the critical pairs in the simplification strategy. Let us first introduce the simple subconvergence criterion:

Definition 4.1 Let R and E be two sets of reductive conditional equations and let E' be the set of conclusions of E . Then E is **simply subconvergent** w.r.t. R if for any conditional equation $(c \Rightarrow l = r) \in E$ and for every test-substitution ϕ such that $l\phi \neq r\phi$, there is a term m such that:

- if $l\phi \succ r\phi$ then : $l\phi (\xrightarrow{R[E']}) o (\xrightarrow{R \cup E'})^* m (\xrightarrow{R \cup E'})^* r\phi$
- if $r\phi \succ l\phi$ then : $r\phi (\xrightarrow{R[E']}) o (\xrightarrow{R \cup E'})^* m (\xrightarrow{R \cup E'})^* l\phi$

Now if all the critical pairs derived from a given system R pass the previous test, then we can be sure that R is ground convergent as the theorem 4.1 shows:

Theorem 4.1 Let R be a set of reductive conditional equations. If all the critical pairs of R are simply subconvergent, then R is ground convergent.

Let us first illustrate the use of theorem 4.1 on a simple example:

Example 4.1 Consider the following unconditional non convergent system R :

$$x + 0 \rightarrow x \quad (1)$$

$$x + s(y) \rightarrow s(x + y) \quad (2)$$

$$(x + y) + z \rightarrow x + (y + z) \quad (3)$$

A test-set for R is $\{0, s(x)\}$. The system is ground convergent by the previous theorem. Consider, for example, a critical pair between associativity and the second rule:

$$s(x + y) + z \rightarrow x + (s(y) + z) \quad (4)$$

and the following instance: $s(s(x) + s(y)) + s(z) \rightarrow s(x) + (s(s(y)) + s(z))$

$$s(s(x) + s(y)) + s(z) \xrightarrow{R} s(s(s(x) + s(y)) + z) \xrightarrow{4} s(s(x) + (s(s(y)) + z))$$

$$s(x) + (s(s(y)) + s(z)) \xrightarrow{R} \xrightarrow{R} s(s(x) + (s(s(y)) + z))$$

Let us now define the subconvergence criterion:

Definition 4.2 Let R and E be two sets of reductive conditional equations. Then E is **subconvergent w.r.t. R** if for any conditional equation $(c \Rightarrow l = r) \in E$ and for every test-substitution ϕ such that $l\phi \neq r\phi$, there is a term m such that:

- if $l\phi \succ r\phi$ then : $l\phi (\xrightarrow{R[E]}) o (\xrightarrow{R \cup E})^* m (\xleftarrow{R \cup E})^* r\phi$
- if $r\phi \succ l\phi$ then : $r\phi (\xrightarrow{R[E]}) o (\xrightarrow{R \cup E})^* m (\xleftarrow{R \cup E})^* l\phi$

We can show that subconvergence, as simple subconvergence, yields ground convergence (the proof is similar):

Theorem 4.2 Let R be a set of reductive conditional equations. If all the critical pairs of R are subconvergent, then R is ground convergent.

We now introduce another test for proving the ground convergence property. In this third test we put more emphasis on the conditional part of the critical pairs. A condition of a rule can help to show that the rule itself is ground convergent:

Definition 4.3 Let R and E be two sets of reductive conditional equations. Then E is **relatively subconvergent w.r.t. R** if for any conditional equation $(c \Rightarrow l = r) \in E$ and for every substitution ϕ obtained by replacing the variable of a test-substitution by new constants (skolemization) and such that $l\phi \neq r\phi$ there is a term m such that:

- if $l\phi \succ r\phi$ then : $l\phi (\xrightarrow{R[c\phi]}) o (\xrightarrow{R \cup E \cup c\phi})^* m (\xleftarrow{R \cup E \cup c\phi})^* r\phi$
- if $r\phi \succ l\phi$ then : $r\phi (\xrightarrow{R[c\phi]}) o (\xrightarrow{R \cup E \cup c\phi})^* m (\xleftarrow{R \cup E \cup c\phi})^* l\phi$

where $c\phi$ means the set of rules $l_i\phi \rightarrow r_i\phi$ such that $l_i = r_i \in c$ and $l_i\phi \succ r_i\phi$. In the following such a rule $l_i\phi \rightarrow r_i\phi$ will be called an **assumption rule**.

If all critical pairs derived from a given system R pass the previous test, then we can be sure again that R is ground convergent (see for instance example 2.2):

Theorem 4.3 Let R be a set of reductive conditional equations. If all the critical pairs of R are relatively subconvergent, then R is ground convergent.

The ground convergence criteria we have designed in this section can set the basis of a completion procedure for deriving a ground convergent system from an initially divergent one: the normalized critical pairs of a given system R which cannot be proved subconvergent by the criteria stated above (definitions 4.1, 4.2, 4.3) are added to the system R to get R' and the ground convergence test is applied to the critical pairs of the new system R' . This process is iterated until a ground convergent to be eventually generated.

We may note here that critical pairs may exist which are ground convergent but cannot be proved so by the previous methods. However, when the initial system is a set R of unconditional equations any critical pair which is essential for completing R to a ground convergent system can be detected by using the definition of test-sets. This procedure is detailed in the full version of the paper.

5 When ground convergence is equivalent to convergence

In this section we give a sufficient condition for the ground convergence property to imply convergence on general terms for rewrite systems compiling equational theories.

Theorem 5.1 Let R be a terminating rewrite system over $T(F, X)$. Suppose that no term of the form $\omega(t, s)$ (where $t, s \in T(F, X)$ and ω is a new symbol not in F) is ground reducible by $R \cup \omega(x, x) \rightarrow x$. Then R is convergent if and only if it is ground convergent.

A direct consequence of this theorem is to give a class of systems where ground convergence is decidable, since convergence of terminating systems is decidable. Note that the criterion of the non existence of ground reducible terms is decidable for left-linear rewrite systems.

References

- [1] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
- [2] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.
- [3] L. Fribourg. A strong restriction of the inductive completion procedure. In *Proceedings 19th International Colloquium on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 105–115. Springer-Verlag, 1986.
- [4] S. Kaplan. Simplifying conditional term rewriting systems : Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295–334, December 1987.
- [5] D. Kapur, P. Narendran, and F. Otto. On ground-confluence of term rewriting systems. *Information and Computation*, 86:14–31, 1990.
- [6] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
- [7] E. Kounalis. Pumping lemmas for tree languages generated by rewrite systems. In *Fifteenth International Symposium on Mathematical Foundations of Computer Science, Banská Bystrica (Czechoslovakia)*. Springer-Verlag, 1990.
- [8] E. Kounalis and M. Rusinowitch. On word problem in Horn logic. volume 308 of *Lecture Notes in Computer Science*, pages 144–160. Springer-Verlag, 1987. See also the extended version published in *Journal of Symbolic Computation*, number 1 & 2, 1991.
- [9] E. Kounalis and M. Rusinowitch. Mechanizing inductive reasoning. In *Proceedings of the AAAI Conference*, pages 240–245, Boston, 1990. AAAI Press and the MIT Press.
- [10] D. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65:182–215, 1985.
- [11] H.T. Zhang and J.-L. Rémy. Contextual rewriting. volume 202 of *Lecture Notes in Computer Science*, pages 46–62, Dijon (France), 1985. Springer-Verlag.
- [12] H.T. Zhang. Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic. PhD thesis. RPI, Troy, 1988.

Solving Divergence in Knuth-Bendix Completion by Enriching Signatures

Muffy Thomas and Phil Watson
Dept. of Computing Science
University of Glasgow

Extended Abstract

1. Introduction

Algebraic specification of abstract data types is a methodology for specifying the behaviour of software systems. A specification consists of a signature and a set of axioms which generate a logical theory; a model of that specification is an algebra with the same signature which satisfies the theory. In equational specification, the axioms are (universally quantified) equations and the theory generated is an equational theory.

Equational reasoning is the process of deriving the consequences of a given system of equations. The simplest way to produce an equational proof that two terms are equal is to keep rewriting subterms of one, using the equations, until it is transformed into the other. The process is more efficient if the equations are considered as *rewrite rules* (rules which represent directed equality). This paradigm is very similar to functional programming; however, in general, rewriting is *nondeterministic* in the sense that no restrictions are placed on the selection of rules to be applied or on the selection of the subterm to be rewritten. Moreover, there is no restriction on overlapping rules.

This generality makes rewriting a very powerful computational paradigm. Our motivation for studying equational reasoning and term rewriting techniques is that if a program or hardware device is described by an equational or algebraic specification, then automated theorem proving systems based on term rewriting can help in showing that it meets its specification. More generally, current uses of rewriting include

- automatic theorem proving of equational and inductive theorems
- solving word problems in universal algebra
- generation of solutions to equations (narrowing)
- prototyping of equational specifications
- synthesis of rewrite rules (which may be regarded as programs, or implementations of more abstract specifications)
- proving properties of specifications such as completeness and consistency

Confluence, termination, and typing are three crucial issues in term rewriting. The *confluence* property ensures that the order of application of rewrite rules is irrelevant, whereas the *termination* property ensures that all sequences of rewrites are well-founded (there are no infinite sequences). The *typing* of terms and operators restricts the applicability of rewrite rules; in order-sorted rewriting [SNGM87], there is a partial

order on the sorts and a substitution x for y is only valid when the sort of x is a subsort of the sort of y .

When a set of rewrite rules is confluent and terminating, then each term has a unique *normal form*: a term which cannot be rewritten. A set of rewrite rules which is confluent and terminating is called *complete*, and makes equality between terms decidable since repeated application of the rules reduces any term to a unique normal form; two terms are equal if and only if they have the same normal forms. The Knuth-Bendix completion algorithm [KB70], given a termination ordering, tests for the confluence property. The algorithm not only tests for confluence, but it also generates a confluent set of rules. It is called a "completion" algorithm because if it terminates, it generates a complete set of rules which can then be used as a decision procedure for equality. The algorithm is not guaranteed to terminate, even when the word problem defined by the given system of equations is decidable. When the completion (semi) algorithm diverges and results in an infinite sequence of confluent rewrite rules, then we only have a semi-decision procedure for the word problem. In practice, many rewriting systems which arise from algebraic specifications are not confluent and the completion algorithm diverges. The only complete solution to the problem of divergence is the use of higher-order rewriting with meta-variables and meta-rules as described in [Ki87]; however, this approach suffers from the difficulties of higher-order rewriting, not least that implementations are not readily available.

This is the problem we attempt to solve: we wish to replace an infinite set of rewrite rules with a finite set satisfying the following two requirements. First, the finite set should preserve the equational theory defined by the given equations, i.e. the finite set should at least be a *conservative extension* [TJ89] of the infinite sequence. (We may also use theorems from the inductive theory when we consider initial algebra semantics.) Note, however, that the finite set may be based on a larger signature than the infinite sequence; the rules in the former may use some sorts which do not occur in the latter. Second, the finite set should be confluent and terminating. Often a finite complete set of rules for an equational theory will not exist in a given signature, and this is why we choose to *enrich* the signature with new sorts.

Our full algorithm is presented in [TW90]. It is too long to give in full here, but the example in section 2 illustrates how it behaves and is in any case more instructive than the list of instructions which make up the algorithm. A brief sketch of the algorithm follows.

We assume that divergence has been detected during an execution of Knuth-Bendix completion. We do not concern ourselves with how this occurs, as this is an interesting problem in its own right [He88]. We assume that an infinite sequence of rules R is being derived by Knuth-Bendix completion, and that each rule takes the form

$$l(x) \rightarrow r(x)$$

where x is a meta-variable instantiated in each rule in R by a term from the language L : the *language of varying parts*. The use of inductive inference techniques to derive L from R is proposed by [TJ89]. Our algorithm takes as input a grammar for L , the *non-varying parts* l, r of the rules, and the initial signature, and outputs a new (order-sorted) signature along with a single rule R' which *exactly generalises* R ; i.e. we have $s \rightarrow t$ using a rule from R in the initial signature iff we have $s \rightarrow t$ using the rule R' in the new signature. We prove that a sufficient condition for our algorithm to succeed is that L be a regular tree language, and characterise this in terms of the more familiar concept of a context-free language. We can also ensure that the new signature is *regular* and *monotonic*, two properties usually regarded as desirable.

2. Example

Consider the set of rules generated by application of the Knuth-Bendix completion algorithm to the rule:

$$R) \quad f(g(f(x))) \rightarrow g(f(x))$$

where we assume a single-sorted signature with no operators apart from $g: T \rightarrow T$, $f: T \rightarrow T$ and a constant symbol $c: T$.

Then the complete set of rules generated is the infinite sequence:

$$\begin{aligned} R1) \quad & f(g(f(x))) \rightarrow g(f(x)) \\ R2) \quad & f(g(g(f(x)))) \rightarrow g(g(f(x))) \\ R3) \quad & f(g(g(g(f(x)))) \rightarrow g(g(g(f(x)))) \\ & \text{etc.} \end{aligned}$$

We use R^∞ to denote this infinite sequence.

It can easily be seen that the rules in R^∞ fall into a clear pattern:

$$f(g^n(f(x))) \rightarrow g^n(f(x)) \quad \text{for any } n > 0.$$

In fact, we might observe that all terms of the form

$$t = g^n(f(x)) \quad \text{for any } n > 0.$$

are qualitatively different from all others; these are exactly the terms for which

$$f(t) \rightarrow t.$$

Note that we cannot generalise R^∞ by the rule

$$f(y) \rightarrow y$$

where y is a variable of sort T . Such a rule does not exactly generalise R^∞ because it is too powerful - it equates terms which have different normal forms under R^∞ . If we add such a rule, then the new rule set is not a conservative extension of the original.

If we were able to define a variable y which could *only* be instantiated by terms of the form $g^n(f(x))$, $n > 0$, then we would be able to replace the infinite sequence by the single rule $f(y) \rightarrow y$. The aim of our algorithm is to define a new sort which contains exactly those terms of the form $g^n(f(x))$, $n > 0$, and to modify the arities of the operators appropriately. The result of applying our algorithm is the following sort structure and operator arities:



$$\begin{array}{ll} g: S \rightarrow S & f: T \rightarrow F \\ g: F \rightarrow S & f: T \rightarrow \text{GLB}(T, F) \\ g: T \rightarrow T & c: T \end{array}$$

The set of terms of sort S is $\{g(f(x)), g(g(f(x))), \dots\}$ and so now the single rule

$$f(y) \rightarrow y$$

with variable y of sort S is a complete rewrite set and a conservative extension of R^∞ . Moreover, the order-sorted signature is monotonic and regular. We note in passing that this new system allows the rewriting of terms of sorts which are incomparable with respect to the sort (inclusion) ordering. In particular we are unable to guarantee the property of sort-decreasingness which is often taken to be a necessary condition for the confluence of a given set rules. Thus this work depends on some method of removing this requirement, e.g. the idea of dynamic sorting [WD89].

3. Conclusions

Term rewriting is a powerful proof tool for algebraic specifications. In practice, many algebraic specifications give rise to infinite sets of complete rules. The algorithm which we have illustrated here is only a part of the full process of transforming an infinite set of rewrite rules R (or more accurately a divergent case of Knuth-Bendix completion) into a finite complete set of rules. If we enrich the original signature Σ in an appropriate way then at least in some cases we arrive at a signature in which at least there exists a complete set of rules which forms a conservative extension of the original set, which may not be true in Σ .

References

[He88]

M. Hermann, Vademecum of Divergent Term Rewriting Systems, CRIN 88-R-082, Centre de Recherche en Informatique de Nancy, 1988.

[Ki87]

H. Kirchner, Schematization of infinite sets of rewrite rules. Application to the divergence of completion processes, in Proc. Rewriting Techniques and Applications, P. Lescanne (ed.), Lecture Notes in Computer Science 256, Springer-Verlag, pp.180-191, 1987.

[KB70]

D. Knuth, P. Bendix, Simple word problems in universal algebra, in J. Leech, ed., Computational Problems in Abstract Algebra, Pergamon Press, 1970.

[SNGM87]

G. Smolka, W. Nutt, J.A. Goguen, J. Meseguer, Order-Sorted Equational Computation, SEKI Report SR-87-14, Universität Kaiserslautern, FRG, 1987.

[TJ89]

M. Thomas, K.P. Jantke, Inductive Inference for Solving Divergence in Knuth-Bendix Completion, Proc. Analogical and Inductive Inference '89, GDR, Lecture Notes in Computer Science 367, Springer-Verlag, 1989.

[TW90]

M. Thomas, P. Watson, Solving Divergence in Knuth-Bendix Completion by Enriching Signatures, Technical Report CSC 90/R15, Glasgow University, 1990.

[WD89]

P. Watson, A.J.J. Dick, Least Sorts in Order-Sorted Term Rewriting, Technical Report TR-CSD-606, Royal Holloway and Bedford New College, University of London, 1989.

Efficient Algebraic Operations on Programs

Neil D. Jones

DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
E-mail: neil@diku.dk

Abstract

A *symbolic version* of an operation on values is a corresponding operation on program texts. For example, symbolic composition of two programs p, q yields a program whose meaning is the (mathematical) composition of the meanings of p and q . Another example is symbolic specialization of a function to a known first argument value. This operation, given the first argument, transforms a two-input program into an equivalent one-input program. Computability of both of these symbolic operations has long been established in recursive function theory [12,16]; the latter is known as Kleene's "s-m-n" theorem, also known as *partial evaluation*.

In addition to computability we are concerned with *efficient* symbolic operations, in particular applications of the two just mentioned to compiling and compiler generation. Several examples of symbolic composition are given, culminating in nontrivial applications to compiler generation [14], [18]. Partial evaluation has recently become the subject of considerable interest [1]. Reasons include simplicity, efficiency and the surprising fact that self-application can be used in practice to generate compilers, and a compiler generator as well.

This paper makes three contributions: First, it introduces a new notation to describe the types of symbolic operations, one that makes an explicit distinction between the types of program texts and the values they denote. This leads to natural definitions of what it means for an interpreter or compiler to be type correct—a tricky problem in a multilanguage context. Second, it uses the notation to give a clear overview of several earlier applications of symbolic computation. For example, it is seen that the new type notation can satisfactorily explain the types involved when generating a compiler by self-applying a partial evaluator. Finally, a number of problems for further research are stated along the way. The paper ends by suggesting Cartesian categorical combinators as a unifying framework in which to study symbolic operations.

1 Introduction

When solving a mathematical problem using a computer one usually thinks first about the subject area: operations on values, functions, matrices, logical formulas, etc.; and only later about how they are to be realized algorithmically. Often, however, a straightforward implementation of a problem statement is too inefficient for practical use, leading to the need for so-called "optimization" to find more efficient computer solutions. Optimization can take many forms. Some of the most powerful speedups result from reformulating the problem—finding a mathematically equivalent restatement that is computationally better. Unfortunately, such methods can be quite subtle, requiring a deep problem understanding that takes years to develop.

An alternative to redoing the mathematics is to let the computer itself find a more efficient implementation of a "naive" algorithm. This leads to the goal of using the computer to transform one program into a new and equivalent but more efficient program.

1.1 High-level Operations in Programming Languages

Programming languages of higher and higher abstraction levels have evolved since the early years of computing (when programming languages were just symbolic codes reflecting the computer's architec-

ture!). Due to higher level basic operations, modern functional languages allow a mathematical style of thinking while programming, for example using function composition, partial function application, set comprehension and pattern matching. This is possible since these mathematical operations are known to give computable results when applied to computable arguments.

On digital computers mathematical operations are specified by textual objects, e.g. programs and their parts, for example expressions. It is well known that many mathematical operations can be faithfully realized by operations on symbolic expressions. A classic example is algebraic symbol manipulation, which abstractly but correctly describes concrete operations on numbers, matrices, etc. The term "symbolic computation" often refers to such manipulations when realized on the computer, but can equally well be interpreted more broadly: to describe the entire theme of this article.

1.2 Operations on Functions and Programs

Two useful operations on (mathematical) functions are:

- *Composition* of f with g , written as $f;g$ or $g \circ f$.
- *Function specialization* of $f(x,y)$, obtaining a one-argument function by "freezing" x to a fixed value, e.g. $x = a$.

Corresponding symbolic operations on programs (assuming for concreteness that they are written in the λ -calculus):

- *Symbolic composition*: suppose f, g are computable by expressions e_f, e_g . Then their composition is computable by expression $\lambda x. e_g(e_f(x))$
- *Partial evaluation*: the specialization of function f to $x = a$ can be realized symbolically as the program $\lambda y. e_f(a, y)$. A well-known version of the same result, in the context of recursive function theory, is Kleene's s-m-n theorem [12,16].

1.3 Efficient Operations on Programs

The symbolic operations above have a common characteristic: while computable, they do not lead to particularly efficient programs. For example the composition program is no faster than just running the two programs from which it is constructed, one after the other. The main theme of this article is the *efficient* implementation of program operations that realize mathematical operations, in particular function composition and specialization. We begin with an example of each.

Composition

Consider the composition $sum \circ squares \circ oneto$, where $oneto(n) = [n, n-1, \dots, 2, 1]$ yields a list of the first n natural numbers, $squares[a_1, a_2, \dots, a_n] = [a_1^2, a_2^2, \dots, a_n^2]$ squares each element in a list, and $sum[a_1, a_2, \dots, a_n] = a_1 + a_2 + \dots + a_n$ sums a list's elements. A straightforward program to compute $sum \circ squares \circ oneto(n)$ is:

```
f(n)      = sum(squares(oneto(n)))
squares(l) = if l = [] then [] else cons(head(l)**2, squares(tail(l)))
sum(l)     = if l = [] then [] else head(l) + sum(tail(l))
oneto(n)   = if n = 0 then [] else cons(n, oneto(n-1))
```

A significantly more efficient and "listless" program [21] for $sum \circ squares \circ oneto$ is:

```
g(n) = if n = 0 then 0 else n**2 + g(n-1)
```

Function Specialization

Consider the following exponentiation program, written as a recursion equation:

```
p(n,x) = if n=0      then 1 else
         if even(n) then p(n/2,x)**2 else x*p(n-1,x)
```

A straightforward specialization of the program above to $n = 5$ is a system of two equations:

```
p5(x) = p(5,x)
p(n,x) = if n=0      then 1 else
         if even(n) then p(n/2,x)**2 else x*p(n-1,x)
```

This corresponds to the trivial λ -calculus construction given above, or the classical proof of Kleene's s-m-n theorem [12]. A better program for $n = 5$ can be got by unfolding applications of the function p and doing all computations involving n and reducing $x*1$ to x , yielding the residual program:

```
p5(x) = x*(x**2)**2
```

1.4 Program Running Times

How can we measure the efficiency increase of the examples just given? Assuming call by value and traditional implementation techniques, a reasonable approximation to program running times on the computer can be obtained as follows. Count 1 for each constant reference, variable reference, test in a conditional or case, function parameter, and base or user-defined function call.

Thus the exponentiation program p above has time estimate:

```
tp(n,x) = 4           if n = 0, else
          15 + tp(n/2,x) if n even, else
          15 + tp(n-1,x)
```

which can be shown to be of order $\log n$. For this specific program and this counting scheme, $t_p(5, x) = 65$. On the other hand the specialized program has running time 7. Similarly, for input n the result of the symbolic composition example takes time $4 + 11 \cdot n$ whereas the "naive" version takes time $15 + 32 \cdot n$.

1.5 Goals of this Article

Our main goals are:

1. To review some nontrivial symbolic operations on programs that have been successfully realized on the computer.
2. To put earlier results into a broader perspective.
3. To list some challenging problems for further study in the area.
4. To propose a more general study of symbolic operations on programs.

Our context of discourse is the following (worth making explicit since it differs from those of many other works about programming languages):

- We consider programs as data objects, and will not discuss their internal structure. (Internal structures and their semantics would be unavoidable if we were discussing *how* to do symbolic composition, etc., but that would be the subject of another paper.)
- Programs' operational behavior will be emphasized, i.e. observable results when applied to first order inputs and outputs.

- We seek a framework for *automatic* and *general* methods, as opposed to one-at-a-time or hand construction of specific program transformations.
- Compiling and compiler generation are of particular interest. This implies a multilanguage context.
- We distinguish between the types of *values* and those of *programs*. For example the number 3 has type *integer*, but Lisp program $(+ 1 2)$ has another type (*integer* *Lisp*, to be explained later).
- Programs will be considered not as untyped but as *polytyped*; for example, a correct compiler translates a source program denoting any expressible type into a target program denoting the same type.

Acknowledgements

This article has been inspired by discussions with (among others) the Topps group at DIKU, John Hughes, John Launchbury, and Alan Mycroft.

2 Definitions and Terminology

We seek a framework to discuss among other things:

- symbolic composition and other operations on programs,
- correctness of compilers, interpreters, and partial evaluators, and
- the types of compilers, interpreters, and partial evaluators.

Compilers and interpreters involve more than one language, so we must first find suitable definitions of programming languages, program meanings, and the values programs manipulate.

Languages will be denoted by Roman letters, e.g. *L* (a default “meta-” or “implementation” language), *S* (a “source language” to be interpreted or compiled) and *T* (a “target language” that is the output of a compiler or partial evaluator). In this paper a program will be assumed to take zero, one, or more first order inputs. An observable result may be a single output, or failure to terminate. We do not consider nondeterminism, communication or other advanced features.

First Order Data

Programs will be treated as data objects, so it is natural to draw both program texts and their input/output data from a common *first order data* domain *D*. Its details are unimportant for this paper, the only essential feature being that any program can be considered as an element of *D*. For example one could follow the well-established Lisp tradition using the set of “S-expressions”¹ for *D*².

Higher Order Values

We will also need higher order values; for example a program’s meaning is often a function from some number of first order inputs to a first order output. It is convenient to assume that the number of a program’s inputs is not fixed in advance. A motivating example: an interpreter takes as input both a program to be interpreted, and *its* input as well. The ability to interpret programs with different numbers of inputs requires that the interpreter itself can accept input sequences of varying lengths³.

We use a universal value domain *V* defined as follows. Appropriate mathematical interpretations are described in [17] and other works on denotational semantics. As is customary we omit summand projections and injections.

¹Defined as the least set such that $D = \text{Atom} \cup D \times D$, where an “atom” is any sequence of one or more letters, digits or characters excluding parentheses, comma and blank.

²In fact any countably infinite set would do, e.g. the natural numbers as in recursive function theory.

³An alternative is to assume that every function has exactly one input, possibly structured. We have tried this but found it to be notationally heavier when defining interpreters, partial evaluators, etc.

$$V = D_{\perp} + (V \rightarrow V)$$

There are three sorts of values: a first order value in D , the undefined value (\perp indicates nontermination), or a function. If a value is a function, its application to an argument yields a value, again of any of these three sorts⁴.

Multiple function arguments are handled by "currying". For example if $f \in V$ is a program meaning with inputs $d_1, \dots, d_n \in D$ then $f d_1 d_2 \dots d_n$ (associated from the left) is the result of applying f to d_1 yielding a function which is then applied to d_2 , etc.

2.1 Programming Languages

Definition 2.1 A programming language is a function $\llbracket - \rrbracket_L : D \rightarrow V$ that associates with each $p \in D$ a value $\llbracket p \rrbracket_L \in V$. The subscript L will be omitted when clear from context.

Intuitively, a language L is identified with its "semantic function" $\llbracket - \rrbracket_L$ on whole programs. All elements p of D are regarded as programs; ill-formed programs can for example be mapped to an everywhere undefined function.

Notational convention. We associate function application from the left and use as few parentheses as possible, for example writing $(\llbracket p \rrbracket_L(d_1))(d_2)$ as $\llbracket p \rrbracket_L d_1 d_2$, or just $\llbracket p \rrbracket d_1 d_2$ if L is understood from context.

Following is enough syntax to describe the result of executing programs on first order data. (It also includes some operationally meaningless syntax, e.g. $3(\llbracket p \rrbracket_L)$.)

Definition 2.2 Abstract syntax for program runs:

$$\begin{aligned} \text{exp} & ::= \text{first-order-data-value} \mid \text{exp}_1 \text{ exp}_2 \mid \llbracket \text{exp} \rrbracket_X \\ \text{first-order-data-value} & ::= \dots \text{ (unspecified) } \end{aligned}$$

Examples for $L = \text{Lisp}$:

$$\begin{aligned} \llbracket (\text{quote ALPHA}) \rrbracket_L & = \text{ALPHA} \\ \llbracket (\text{lambda } (x) (+ x x)) \rrbracket_L 3 & = 6 \end{aligned}$$

2.2 Data and Program Types

We now extend the usual concept of type.

Definition 2.3 The Abstract syntax of a type t :

$$t: \text{type} ::= \underline{\text{type}}_X \mid \text{firstorder} \mid \text{type} \times \text{type} \mid \text{type} \rightarrow \text{type}$$

Type *firstorder* describes values in D , i.e. S-expressions, and function types and products are as usual. For each language X and type t we have a type constructor, written \underline{t}_X and meaning the type of all X -programs which denote values of type t . For example, atom ALPHA has type *firstorder*, and Lisp program (quote ALPHA) has type $\underline{\text{firstorder}}_{\text{Lisp}}$.

Type Deduction Rules Figure 1 contains some type deduction rules sufficient to describe program runs, i.e. evaluations of closed expressions.

⁴Note that this definition allows a function to have varying numbers of arguments.

$\frac{exp_1 : t_2 \rightarrow t_1, \quad exp_2 : t_2}{exp_1 exp_2 : t_1}$	$\frac{exp : t_X}{[exp]_X : t}$
$\frac{}{firstordervalue : firstorder}$	$\frac{exp : t_X}{exp : firstorder}$

Figure 1: Some Type Inference Rules for Closed Expressions

Remark An object p of type t_X is a program text and thus *in itself* a value in D , i.e. t_X denotes a subset of D . On the other hand, p 's denotation $[p]_X$ may be any value in V , for example a higher order function.

Semantics of Types

Definition 2.4 The meaning of type expression t is $\llbracket t \rrbracket$ defined as follows:

$$\begin{aligned} \llbracket firstorder \rrbracket &= D \\ \llbracket t_1 \rightarrow t_2 \rrbracket &= \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket \\ \llbracket t_1 \times t_2 \rrbracket &= \{(t_1, t_2) \mid t_1 \in \llbracket t_1 \rrbracket, t_2 \in \llbracket t_2 \rrbracket\} \\ \llbracket t_X \rrbracket &= \{p \in D \mid [p]_X \in \llbracket t \rrbracket\} \end{aligned}$$

Polymorphism

We will also allow *polymorphic* type expressions to be written containing *type variables* $\alpha, \beta, \gamma, \dots$. Such a polymorphic type will always be understood relative to a substitution mapping type variables to type expressions without variables. The result of applying the substitution is called a *substitution instance*.

Program Equivalence It is important to be able to say when two programs $p, q \in D$ are computationally equivalent. In recent years two views have developed, *semantic equivalence* and *operational equivalence*. Both concepts make sense in our framework, as defined by:

Definition 2.5 Let $p, q \in D$. Then

- p and q are *semantically equivalent* if $[p] = [q]$
- p and q are *operationally equivalent* if $[p] \approx [q]$, where for $f, g \in D$ we define $f \approx g$ to mean that for all $d_1, \dots, d_n \in D$ and $d \in D_\perp$,

$$fd_1 \dots d_n = d \text{ if and only if } gd_1 \dots d_n = d$$

The first definition is the simpler of the two, but a case can be made that the second definition is likely to be more relevant in practice. The reason is that the first definition requires verifying equality between two elements of a semantic function domain. This can be a tricky task, and impossible if the semantic function $[-]$ is not fully abstract.

The second definition is a version of the more general operational equivalence studied by Plotkin, Milner, and others, limited to first order applicative contexts. It only involves assertions that can in principle be verified by running the program on first order inputs and observing its first order outputs or nontermination behavior.

2.3 Some Problems for Study

1. Find a suitable model theory for these types (domains, ideals, ...). The semantics above uses ordinary sets, but for computational purposes it is desirable that the domains be algebraic, and the values which manipulate programs should only range over computable elements.

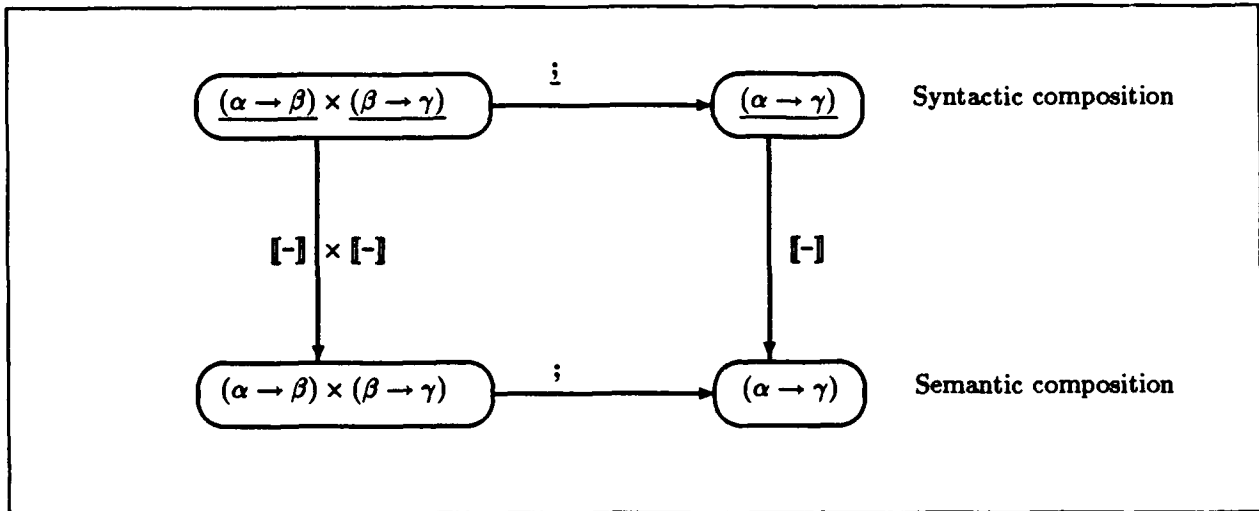


Figure 2: Symbolic Composition

2. Find a sound set of type rules for a familiar language that are sufficiently powerful to describe the types of compilers and interpreters (more on this in section 5.3).
3. Figure 1 contains no rules for deducing that any programs at all have types of form \underline{t}_L . Problem: for a fixed programming language, find type inference rules appropriate for showing that given programs have given types. (More will be said on this in a later section.)

3 Efficient Symbolic Composition

Symbolic composition can be described as commutativity of the diagram in Figure 2, where α, β, γ range over all types. We now list several examples of symbolic composition, and discuss what is saved computationally.

3.1 Vector Spaces and Matrix Multiplication

An $m \times n$ matrix M over (for example) the real numbers R determines a linear transformation $[M] : R^n \rightarrow R^m$, so $[M]\vec{v}$ is a vector $\vec{w} \in R^m$ if $\vec{v} \in R^n$. If M, N are respectively $m \times n$ and $n \times p$ matrices and $M \cdot N$ their matrix product, then $[M \cdot N] : R^p \rightarrow R^m$ and we have

$$[M \cdot N](\vec{w}) = [M]([N](\vec{w}))$$

Assuming $m = n = p$, the composition $[M]([N](\vec{w}))$ can be computed in either of two ways:

- by applying first N and then M to \vec{w} (time $2n^2$), or
- by first multiplying M and N (time n^3 by the naive algorithm), and applying the result to \vec{w} (time n^2)

It may be asked: what if anything has been saved? The answer is: nothing, if the goal is only to transform a single vector, since the second time always exceeds the first. There is, however, a net saving if more than n vectors are to be transformed since the matrix multiplication need only be computed once.

The moral: so familiar an operation as matrix multiplication can be thought of as symbolic composition; and composition can save computational time.

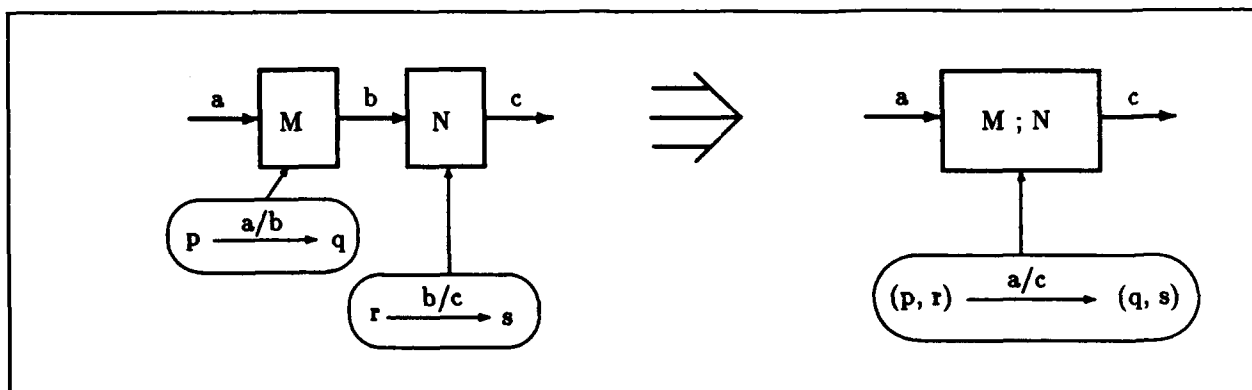


Figure 3: Composition of Finite State Transducers

3.2 Finite State Transducers

Suppose one is given two finite state transducers: M which transforms input strings from Σ^* into Δ^* , and N which transforms strings from Δ^* into Γ^* . It is well known that these can be combined into a single finite state transducer $M;N$ that directly transforms strings from Σ^* into Γ^* without ever producing any intermediate strings. The construction is straightforward: a state of $M;N$ is a pair of states, one from M and one from N . Transitions in $M;N$ are defined as indicated in the diagram of Figure 3.

3.3 Derivors

A derivor (terminology taken from the ADJ group [9]) is a translation from one many-sorted algebra to another, usually involving a change of signature (a signature morphism is an example). Derivors must be *compositional*: the translation of a composite term is a combination of the translations of its components. The requirement of compositionality is very natural for *denotational semantics* [17].

Derivors over term algebras can also be thought of as finite state transducers that have been extended to operate on trees (strings are a special case). Derivors can also be composed symbolically in a way very analogous to finite transducers. For a small example, consider the three term algebras $T(A), T(B), T(C)$ over signatures given in Figure 4.

Using a natural denotational notation we now define two derivors. One is $[-]_{AB}$ from $T(A)$ to $T(B)$:

$$\begin{aligned} [0]_{AB} &= \text{zero} \\ [1]_{AB} &= \text{one} \\ [a_1 + a_2]_{AB} &= [a_1]_{AB} \text{ plus } [a_2]_{AB} \\ [a_1 - a_2]_{AB} &= [a_1]_{AB} \text{ plus (minus } [a_2]_{AB}) \end{aligned}$$

and the second is $[-]_{BC}$ from $T(B)$ to $T(C)$:

$$\begin{aligned} [\text{zero}]_{BC} &= \text{push}(0) \\ [\text{one}]_{BC} &= \text{push}(1) \\ [b_1 \text{ plus } b_2]_{BC} &= [b_1]_{BC} ; [b_2]_{BC} ; \text{add} \\ [\text{minus } b]_{BC} &= [b]_{BC} ; \text{negate} \end{aligned}$$

Their composition $[-]_{AC}$ from $T(A)$ to $T(C)$ is obtained by applying $[-]_{BC}$ to the right side of every rule defining $[-]_{AB}$. This process yields a more efficient version in which no B terms are ever constructed:

$$\begin{aligned} [0]_{AC} &= \text{push}(0) \\ [1]_{AC} &= \text{push}(1) \\ [a_1 + a_2]_{AC} &= [a_1]_{AC} ; [a_2]_{AC} ; \text{add} \\ [a_1 - a_2]_{AC} &= [a_1]_{AC} ; ([a_2]_{AC} ; \text{negate}) ; \text{add} \end{aligned}$$

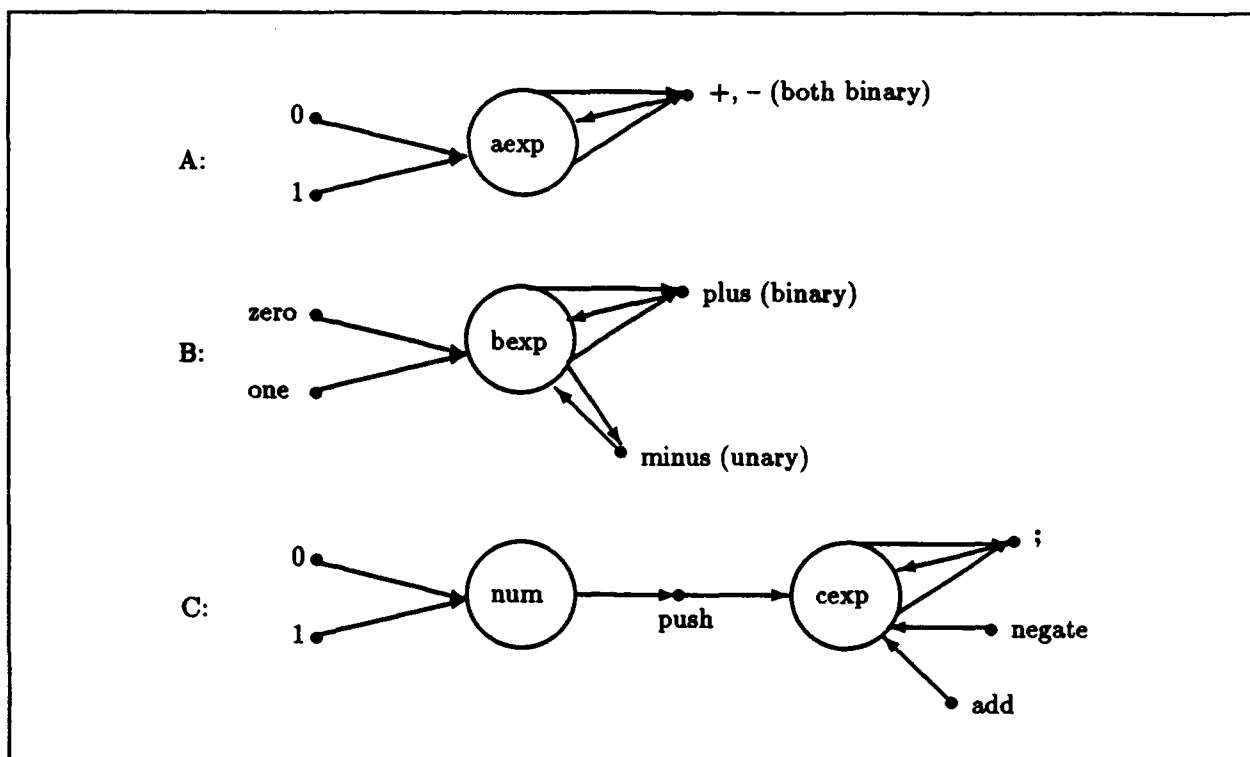


Figure 4: Three Signatures

3.4 Attribute Coupled Grammars

These define term-to-term mappings; typically the terms are abstract syntax trees. Attribute coupled grammars have much in common with derivors, but extend them significantly in several ways. One is that multiple attributes are allowed, and another is that attribute values may flow both up and down a term. (Derivors correspond to the special case of only one synthesized attribute.) Another extension is to allow nontrivial computation and not just the assembly of subtrees.

Ganzinger and Giegerich show in [7] that attribute coupled grammars can be composed symbolically (too complex an operation to illustrate here). Their motivation was to develop a general framework for automatic pass composition and decomposition in the context of multipass compilers.

3.5 Functional programs

We saw an example in the introduction of composing functional programs, with a significant increase in speed. Turchin reports in [20] the transformation of a two pass program into a one pass equivalent. This was followed by Wadler's "listless transformer" [21], more clearly described and for a more traditional lazy functional language. Both approaches are implemented but semiautomatic - they often give good results when they terminate, but are complex and not guaranteed to terminate. Wadler's "treeless transformer" is an advance [22] in that it always terminates, often with significant efficiency improvements, but for a very limited class of programs.

While the state of the art has advanced rapidly, there is still much room for improvement in the symbolic composition of functional programs.

3.6 Some Problems for Study

1. Find a generalization of attribute coupled grammars with a simpler composition algorithm, or with greater computational powers.

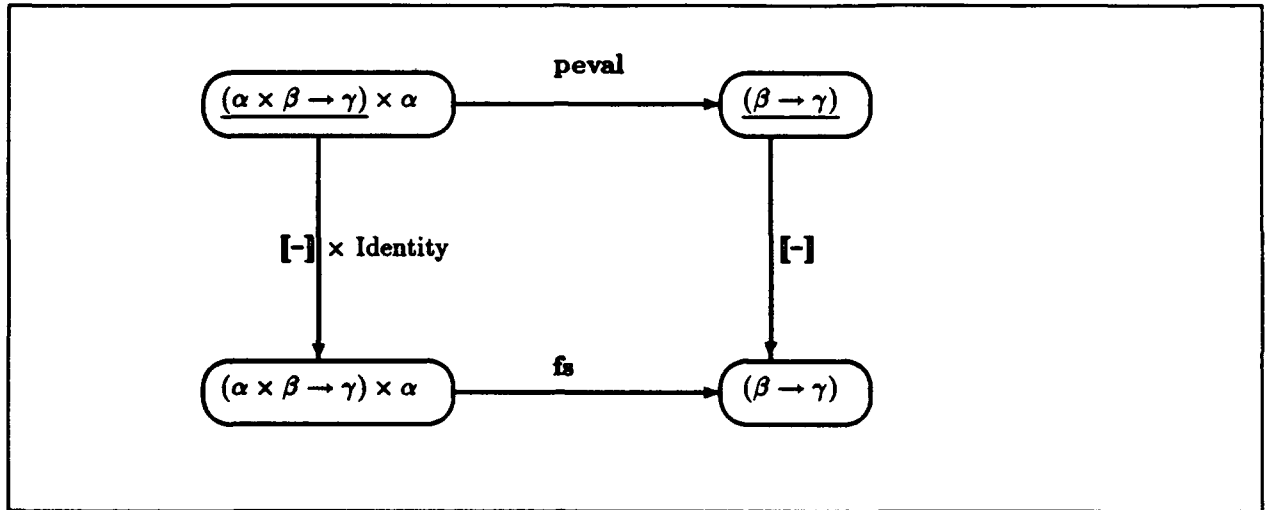


Figure 5: Function Specialization and Partial Evaluation

2. Find a more general and powerful method for symbolic composition of functional programs.

4 Symbolic Function Specialization = Partial Evaluation

4.1 Function Specialization

The result of *specializing* (also called *restricting*) a two argument function $f(x, y) : \alpha \times \beta \rightarrow \gamma$ to a fixed value $x = a$ is the function

$$f|_{x=a}(y) = f(a, y)$$

for all y . Function specialization thus has polymorphic type

$$fs : (\alpha \times \beta \rightarrow \gamma) \times \alpha \rightarrow (\beta \rightarrow \gamma)$$

Partial Evaluation Partial evaluation is the symbolic operation corresponding to function specialization. Given a *program* for f and a value $x = a$, a partial evaluator yields a program to compute $f|_{x=a}$. Using *peval* to denote the partial evaluation function, its correctness is expressed by commutativity of the diagram in Figure 5. Partial evaluation has polymorphic type

$$peval : (\alpha \times \beta \rightarrow \gamma) \times \alpha \rightarrow (\beta \rightarrow \gamma)$$

A partial evaluator deserves its name because, when given a program and incomplete input, it will do part of the evaluation the program would do on complete input. A partial evaluator is in essence a *program specializer*: given a program and the values of part of its input data, it yields another program which, given its remaining input, computes the same value the original program gives on all its input. In still other words, a partial evaluator is a computer realization of Kleene's s-m-n theorem [12,16].

The traditional proofs of Kleene's s-m-n theorem do not take efficiency into account, yielding trivial specialized programs. Efficiency is very important in applications though, so partial evaluation may be regarded as the quest for efficient implementations of the s-m-n theorem.

4.2 Definition of Partial Evaluation

The description of *peval* can be both simplified and generalized by writing the functions involved in *curried* form. (Recall that $\alpha \times \beta \rightarrow \gamma$ is isomorphic to $\alpha \rightarrow (\beta \rightarrow \gamma)$.) This gives *peval* a new polymorphic type:

$$peval : \alpha \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

which is an instance of the more general polymorphic type:

$$\text{peval} : \underline{\rho} \rightarrow \underline{\sigma} \rightarrow \underline{\rho} \rightarrow \underline{\sigma}$$

Maintaining our emphasis on observable values, we will require ρ to be a first order type (i.e. *base* or an *a type* \underline{t}_L). A final restriction: *peval* should be computable (as in [12,16]) so we require $\text{peval} = \llbracket \text{mix} \rrbracket$ for some program *mix*. (The name *mix* comes from [5].) Finally, we come to the definition:

Definition 4.1 The *mix* equation. Program *mix* $\in D$ is a partial evaluator if for all *p*, *a* $\in D$,

$$\llbracket p \rrbracket a \approx \llbracket \llbracket \text{mix} \rrbracket p a \rrbracket$$

By definition of \approx

$$\llbracket p \rrbracket d_1 d_2 \dots d_n = d$$

if and only if

$$\llbracket \llbracket \text{mix} \rrbracket p d_1 \rrbracket d_2 \dots d_n = d$$

The same concept may be generalized to differing input, output and implementation languages, but we will only need one-language partial evaluators.

4.3 Some Techniques for Partial Evaluation

Partial evaluators that have been successfully used for compiling and compiler generation include [8], [2], [3] and [11]. The *mix*-time techniques used there include: applying base functions to known data; *unfolding* function calls; and possibly creating one or more *specialized program points*. A program point comes from the source program (perhaps the name of a user-defined function) and is specialized to known data values computable from the known input d_1 .

To illustrate these techniques, consider the well-known example of Ackermann's function:

```
a(m,n) =   if m=0 then n+1 else
           if n=0 then a(m-1,1)
           else a(m-1,a(m,n-1))
```

Computing $a(2,n)$ involves recursive evaluations of $a(m,n')$ for $m = 0, 1$ and 2 , and various values of n' . The partial evaluator can evaluate $m=0$ and $m=1$ for the needed values of m , and function calls of form $a(m-1, \dots)$ can be unfolded (i.e. replaced by the right side of the recursive equation above, after the appropriate substitutions).

By these techniques we can specialize function *a* to the values of $m = 1$ and 2 , yielding the residual program:

```
a2(n) = if n=0 then 3 else a1(a2(n-1))
a1(n) = if n=0 then 2 else a1(n-1)+1
```

This program performs less than half as many arithmetic operations as the original since all tests on and computations involving *m* have been removed. The example is admittedly pointless for practical use due to the enormous growth rate of Ackermann's function, but it well illustrates some general optimization techniques.

5 Compilers, Interpreters and Their Types

Let L , S and T be programming languages (respectively source, implementation and target languages). Our default language is L , and we often abbreviate $[p]_L$ to $[p]$.

Definition 5.1

1. An interpreter for S written in L is a member *int* of the set

$$\boxed{\begin{array}{c} S \\ L \end{array}} \stackrel{\text{def}}{=} \{ \text{int} \in D \mid \forall s \in D. [s]_S \approx [\text{int}]_L s \}$$

2. An S -to- T -compiler written in L is a member *comp* of the set

$$\boxed{\begin{array}{c} S \rightarrow T \\ L \end{array}} \stackrel{\text{def}}{=} \{ \text{comp} \in D \mid \forall s \in D. [s]_S \approx [[\text{comp}]_L s]_T \}$$

5.1 The Computational Overhead Caused by Interpretation

On the computer, interpreted programs are often observed to run slower than compiled ones. To explain why, let $t_p(d_1 \dots d_n)$ denote the time required to calculate $[p]_L d_1 \dots d_n$. An interpreter's basic cycle is usually first syntax analysis: a series of tests to determine the main operator of the current expression to be evaluated; then evaluation of necessary subexpressions by recursive calls to its evaluation function; and finally, actions to evaluate the expression's main operator, e.g. to subtract 1 or to look up the current value of a variable. It is very common in practice that the running time of an interpreter *int* on input $(p \ b)$ satisfies

$$a \cdot t_p(d) \leq t_{\text{int}}(p \ d)$$

for all d , where a is a constant. (In this context "constant" means: a is independent of d , but it may depend on p .) In our experiments a is often around 10 for small source programs.

This is typical of many interpreters in our experience: an interpreted program runs slower than one which is compiled; and the difference is a constant factor, large enough to be worth reducing for practical reasons, and depending on the size of the program being interpreted. Clever use of data structures (hash tables, binary trees, etc) can make a grow slowly as a function of p 's size.

5.2 Can an Interpreter be Typed?

Suppose we have an interpreter *up* for language L , and written in the same language—in other words a "universal program" or "self-interpreter". Can *up* be proven type correct? By definition *up* must satisfy

$$[p] \approx [\text{up}] \ p$$

for any L -program p . Consequently as p ranges over all L -programs, $[\text{up}] \ p$ can take on *any* program-expressible type. A difficult question arises: is it possible to define the type of *up* nontrivially?

This question does not arise at all in classical computability theory since there is only one data type—the natural numbers, and all programs denote functions on them. On the other hand, computer scientists are unwilling to code all data as numbers and so demand programs with varying input, output and data types.

A traditional Computer Science response to this problem has been to write an interpreter in an *untyped* language, e.g. Lisp or Scheme. This has the disadvantage that it is hard to verify that the interpreter correctly implements the type system of its input language (if any). The reason is that there are two classes of possible errors: those caused by errors in the program being interpreted, and those caused by a badly written interpreter. Without a type system it is difficult to distinguish the one class of interpret-time errors from the other.

An alternative approach is to use the types t_x as described earlier. Given this notation and a sufficiently strong type system, we conjecture that it is possible to prove that interpreters, compilers and partial evaluators properly deal with the types of their program inputs and outputs.

5.3 Well-typed Language Processors

Given a source S -program denoting a value of some type t , an S -interpreter should return a value whose type is t . From the same source program, a compiler should yield a target language program whose T -denotation is identical to its source program's S -denotation. This agrees with daily experience—a compiler is a meaning-preserving program transformation, insensitive to the type of its input program (provided only that it is well-typed). Analogous requirements apply to partial evaluators.

Interpreters A well-typed interpreter is required to have many types: one for every possible input program type. Thus to satisfy these definitions we must dispense with type *unicity*, and allow the type of the interpreted program not to be uniquely determined by its syntax.

Compilers Compilers must satisfy an analogous demand. One example: $\lambda x.x$ has type $\underline{t}_L \rightarrow \underline{t}_L$ for all types t . It is thus a trivial but well-typed compiler from L to L .

Partial Evaluators A well-typed partial evaluator can be applied to any program p accepting at least one first order input, together with a value d_1 for p 's first input. Suppose p has type $\underline{\rho} \rightarrow \underline{\sigma}$ where ρ is first order and $d_1 \in \llbracket \rho \rrbracket$. Then $\llbracket \text{mix} \rrbracket p \ d_1$ is a program whose result type is σ , the type of $\llbracket p \rrbracket d_1$.

Definition 5.2 ⁵

1. Interpreter $\text{int} \in \begin{matrix} S \\ L \end{matrix}$ is well-typed if $\llbracket \text{int} \rrbracket_L$ has type $\forall \delta. \underline{\delta}_S \rightarrow \delta$.
2. Compiler $\text{comp} \in \begin{matrix} S \rightarrow T \\ L \end{matrix}$ is well-typed if $\llbracket \text{comp} \rrbracket_L$ has type $\forall \alpha. \underline{\alpha}_S \rightarrow \underline{\alpha}_T$.
3. A partial evaluator mix is well-typed if $\llbracket \text{mix} \rrbracket$ has type $\forall \rho. \forall \sigma. \underline{\rho} \rightarrow \underline{\sigma} \rightarrow \rho \rightarrow \underline{\sigma}$, where ρ ranges over first order types.

Remark The definition of a well-typed interpreter assumes that all S -types are also L -types (at least, observably so). Thus it does not take into account the possibility of encoding S -values, as is often seen in computing practice.

5.4 Problems for Study

1. Verify type correctness of a simple interpreter with integer and boolean data
2. Formulate a set of type inference rules for a full language, e.g. the λ -calculus, that is sufficiently general to verify type correctness of a range of compilers, interpreters and partial evaluators

6 Partial Evaluation and Compiler Generation

6.1 The Futamura Projections

Suppose we are given an interpreter int for some language S , written in L . Letting source be an S -program, a compiler from S to L produces an equivalent L -program target , meaning that $\llbracket \text{source} \rrbracket_S$ and $\llbracket \text{target} \rrbracket_L$ are the same value in V . Recall the mix equation:

$$\llbracket p \rrbracket a \approx \llbracket \llbracket \text{mix} \rrbracket p a \rrbracket$$

for all $p, a \in B$. The following three equations are the so-called "Futamura projections" [6,5]. They assert that given a partial evaluator mix and an interpreter int , one may compile programs, generate compilers and even generate a compiler generator.

⁵To say, for example, that int has type $\forall \delta. \underline{\delta}_S \rightarrow \delta$ means that int has type $\underline{t}_S \rightarrow t$ for every type t .

$\llbracket \text{mix} \rrbracket \text{int source}$	$=$	target	
$\llbracket \text{mix} \rrbracket \text{mix int}$	$=$	compiler	
$\llbracket \text{mix} \rrbracket \text{mix mix}$	$=$	cogen	a compiler generator

6.1.1 Compilation

Program **source** from the interpreted language **S** has been translated to program **target**. It is natural that **target** is in language **L** since it is a specialized version of **L-program int**. It is easy to verify that the target program is faithful to its source using the definitions of interpreters, compilers and the mix equation:

$\llbracket \text{source} \rrbracket_S$	\approx	$\llbracket \text{int} \rrbracket \text{source}$	definition of an interpreter
	\approx	$\llbracket \llbracket \text{mix} \rrbracket \text{int source} \rrbracket_T$	mix equation
	$=$	$\llbracket \text{target} \rrbracket_T$	definition of target

Why bother? The reason is that it is usually considerably faster to run **target** directly on a **T-machine** than it would be to run **source** on an **L-machine**, due to the removal of the interpretive overhead discussed in section 5.1.

6.1.2 Compiler Generation

We have just seen that **mix** can be used to compile from one language to another, given an interpreter for the source language. The second Futamura projection says that it can also generate stand-alone compilers. Verification that program **compiler** translates source programs into equivalent target programs is also straightforward:

target	\approx	$\llbracket \text{mix} \rrbracket \text{int source}$	definition of target
	\approx	$\llbracket \llbracket \text{mix} \rrbracket \text{mix int} \rrbracket \text{source}$	mix equation
	$=$	$\llbracket \text{compiler} \rrbracket \text{source}$	definition of a compiler

6.1.3 Generating a Compiler Generator

Finally, we can see that **cogen** transforms interpreters into compilers by the following:

compiler	\approx	$\llbracket \text{mix} \rrbracket \text{mix int}$	definition of target
	\approx	$\llbracket \llbracket \text{mix} \rrbracket \text{mix mix} \rrbracket \text{int}$	mix equation
	$=$	$\llbracket \text{cogen} \rrbracket \text{int}$	definition of cogen

6.2 Self-Application and Types

Definitions involving self-application often (and rightly) cause concern as to their well-typedness. We show here that natural types for **target** and **compiler** can be deduced from the few type rules given earlier (and even **cogen**, which we omit because of the deduction's complexity). Recall their polymorphic types:

Type of source :	$\underline{\delta}_S$
Type of int :	$\forall \delta. \underline{\delta}_S \rightarrow \delta$
Type of compiler :	$\forall \delta. \underline{\delta}_S \rightarrow \underline{\alpha}_T$
Type of mix :	$\forall \rho. \forall \sigma. \underline{\rho} \rightarrow \sigma \rightarrow \rho \rightarrow \underline{\sigma}$ where ρ is first order

First Futamura Projection

We wish to find the type of **target** = $\llbracket \text{mix} \rrbracket \text{int source}$. The following deduction assumes that program **source** has type $\underline{\delta}_S$ and concludes that the target program has type $\underline{\delta} = \underline{\delta}_L$, i.e. that it is an **L** program of the right type. The deduction uses only the rules of Figure 1 and generalization of polymorphic variables.

$$\begin{array}{c}
\text{mix} : \underline{\underline{\rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma}} \\
\hline
\llbracket \text{mix} \rrbracket : \underline{\underline{\rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma}} \\
\hline
\llbracket \text{mix} \rrbracket : \underline{\underline{\delta_S \rightarrow \delta \rightarrow \delta_S \rightarrow \delta}} \quad \text{int} : \underline{\underline{\delta_S \rightarrow \delta}} \\
\hline
\llbracket \text{mix} \rrbracket \text{int} : \underline{\underline{\delta_S \rightarrow \delta}} \quad \text{source} : \underline{\underline{\delta_S}} \\
\hline
\llbracket \text{mix} \rrbracket \text{int source} : \underline{\underline{\delta}}
\end{array}$$

Second Futamura Projection

The previous deduction showed that $\llbracket \text{mix} \rrbracket \text{int}$ has the type of a compiling *function*, though it is not a compiler *program*. We now wish to find the type of $\text{compiler} = \llbracket \text{mix} \rrbracket \text{mix int}$. It turns out to be notationally simpler to begin with a program p of more general type $\underline{\underline{\alpha \rightarrow \beta}}$ than that of int .

$$\begin{array}{c}
\text{mix} : \underline{\underline{\rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma}} \\
\hline
\llbracket \text{mix} \rrbracket : \underline{\underline{\rho \rightarrow \sigma \rightarrow \rho \rightarrow \sigma}} \\
\hline
\llbracket \text{mix} \rrbracket : \underline{\underline{\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta}} \quad \text{mix} : \underline{\underline{\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta}} \\
\hline
\llbracket \text{mix} \rrbracket \text{mix} : \underline{\underline{\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta}} \quad p : \underline{\underline{\alpha \rightarrow \beta}} \\
\hline
\llbracket \text{mix} \rrbracket \text{mix } p : \underline{\underline{\alpha \rightarrow \beta}}
\end{array}$$

Some Interesting Substitution Instances

By the second Futamura projection, $\text{compiler} = \llbracket \text{mix} \rrbracket \text{mix int}$. The type of $p = \text{int}$ is $\underline{\underline{\delta_S \rightarrow \delta}}$, an instance of the type assigned to p above. By the same substitution we have $\llbracket \text{compiler} \rrbracket : \underline{\underline{\delta_S \rightarrow \delta}}$. Furthermore δ was chosen arbitrarily, so $\llbracket \text{compiler} \rrbracket : \forall \delta. \underline{\underline{\delta_S \rightarrow \delta}}$ as desired.

The Type of a Compiler Generator. Even cogen can be given a type, namely

$$\underline{\underline{\underline{\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta}}}$$

by exactly the same technique; but the tree is too complex to display. One substitution instance of cogen 's type is the conversion of an interpreter's type into that of a compiler.

The type of $\llbracket \text{cogen} \rrbracket$ is clearly $\underline{\underline{\alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta}}$. This is just an instance of the type of the identity function(!) but with some underlining. It is substantially different, however, in that it describes program generation. Specifically

1. $\llbracket \text{cogen}' \rrbracket$ transforms a two input program p into another program $p\text{-gen}$, such that for any $d_1 \in D$
2. $p' = \llbracket p\text{-gen} \rrbracket d_1$ is a program
3. which for any $d_2 \in D$ computes

$$\llbracket p' \rrbracket d_2 \approx \llbracket p \rrbracket d_1 d_2$$

Ershov called program *p-gen* the *generating extension* of *p*—a logical name since it is a program that generates specialized versions of *p*, when given values d_1 of its first input.

One could even argue that the function $\llbracket \text{cogen} \rrbracket$ realizes an *intensional version of currying*, one that works on program texts instead of on functions⁶. To follow this, the type of $\llbracket \text{cogen} \rrbracket$ has as a substitution instance

$$\llbracket \text{cogen} \rrbracket : \alpha \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

In most higher order languages it requires only a trivial modification of a program text with type $(\alpha \rightarrow (\beta \rightarrow \gamma))$ to obtain a variant with type $(\alpha \times \beta \rightarrow \gamma)$, and with identical computational complexity. So a variant *cogen'* could be easily constructed that would first carry out this modification on its program input, and then run *cogen* on the result. The function computed by *cogen'* would be of type:

$$\llbracket \text{cogen}' \rrbracket : \alpha \times \beta \rightarrow \gamma \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

6.3 Efficiency Issues

6.3.1 Self-Application can Improve Efficiency

A variety of partial evaluators satisfying all the above equations have been constructed ([8,3,11] contain more detailed discussions). Compilation, compiler generation and compiler generator generation can each be done in two ways, to wit:

$$\begin{aligned} \text{target} &= \llbracket \text{mix} \rrbracket \text{int source} \\ &= \llbracket \text{compiler} \rrbracket \text{source} \\ \\ \text{compiler} &= \llbracket \text{mix} \rrbracket \text{mix int} \\ &= \llbracket \text{cogen} \rrbracket \text{int} \\ \\ \text{cogen} &= \llbracket \text{mix} \rrbracket \text{mix mix} \\ &= \llbracket \text{cogen} \rrbracket \text{mix} \end{aligned}$$

Although the exact timings vary according to the partial evaluator and the implementation language *L*, in each case the second way is often about 10 times faster than the first. A less machine dependent and more intrinsic efficiency measure follows.

6.3.2 Optimality of Partial Evaluation

For practical purposes the trivial partial evaluation given by the traditional *s-m-n* construction (e.g. as illustrated in the introduction) is uninteresting; in effect it would yield a target program of the form “apply the interpreter to the source program text and its input data”. Ideally, *mix* should remove *all computational overhead* caused by interpretation.

How can we meaningfully assert that a partial evaluator is “good enough”? Perhaps surprisingly, a machine-independent answer can be given. This answer involves the *mix* equation and a self-interpreter

$$\text{up} \in \begin{bmatrix} L \\ L \end{bmatrix}$$

For any program *p*

$$\llbracket p \rrbracket \approx \llbracket \text{up} \rrbracket p \approx \llbracket \llbracket \text{mix} \rrbracket \text{up} p \rrbracket$$

so $p' = \llbracket \text{mix} \rrbracket \text{up} p$ is an *L*-program equivalent to *p*. This suggests a natural goal: that *p'* be at least as efficient as *p*. Achieving this goal implies that *all computational overhead* caused by *up*'s interpretation has been removed by *mix*.

⁶The well-known “curry” isomorphism on functions is:

$$\text{curry} : (\alpha \rightarrow (\beta \rightarrow \gamma)) \simeq (\alpha \times \beta \rightarrow \gamma)$$

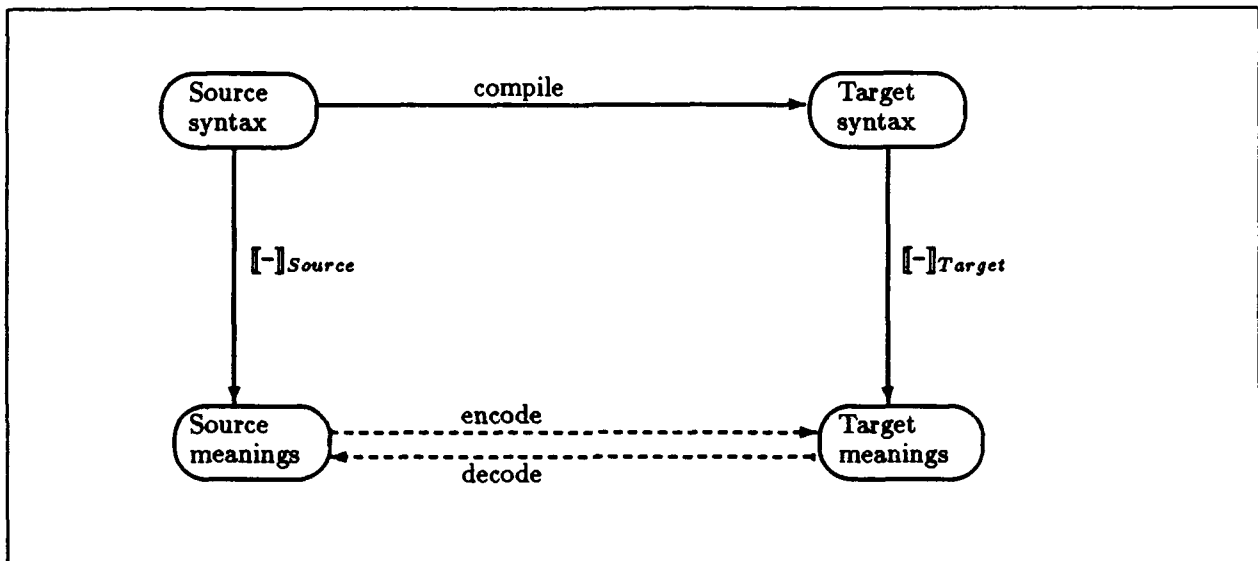


Figure 6: Morris and ADJ Advice on Compiler Construction

Definition 6.1 *Mix is optimal provided $p \approx \llbracket \text{mix} \rrbracket \text{ up } p$ for all $p \in B$.*

We have satisfied this criterion on the computer for several partial evaluators for various self-interpreters (e.g. [11]). In each case the residual program is identical to p (up to variable renaming).

7 Symbolic Composition and Compiler Generation

We describe two algebraic approaches to compiler generation in which symbolic composition plays a central role.

7.1 The Morris and ADJ Advice on Compiler Construction

This does not involve symbolic composition, but we mention it briefly to put the next two examples into perspective. The first “advice” by Morris [13] was to express the translation algebraically, regarding source program syntax, target program syntax, and source and target meanings as many-sorted algebras, and regarding both the semantic and compilation functions as homomorphisms. Derivators were used for the necessary signature changes.

An important consequence is that compiler correctness can be proven by purely algebraic means. The ADJ group [10] carried Morris’ advice further, extending his results to a more powerful source language and providing a more purely algebraic correctness proof. Correctness was proven by showing commutativity of versions of the diagram in Figure 6.

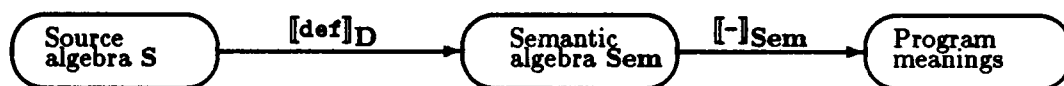
The Morris/ADJ algebraic approaches provide a rich framework for constructing and manipulating programs, program pieces and their meanings, and does indeed provide a way to prove compilers correct. However their advice is less suitable if the goal is *compiler generation* for a variety of source languages. One problem is that it requires a large amount of work to define the appropriate algebras and semantic functions - work that must be redone from scratch for every new language or compiler. Further, devising “encode” or “decode” functions that work is a decidedly nontrivial task, and one hard to do systematically.

7.2 Denotational Semantics

Denotational semantics provides a systematic framework for defining semantics of a wide range of programming languages. A denotational language definition specifies program phrase meanings in two steps

with the aid of a single, universal "semantic language" which we will call **Sem** (quite often the λ -calculus⁷). The effect is that a program's meaning is *by definition* the meaning of the semantic language expression into which it is mapped. Clearly **Sem** should be a broad spectrum language, suitable for assigning meanings to a wide variety of programming languages.

The classical "denotational assumption" is just compositionality, so the mapping from source language terms can be given by a derivor from the source syntax algebra **S** to the semantic algebra **Sem**. To describe this more formally, let **D** be the language of derivors, so derivor **def** defines a mapping $[[\mathbf{def}]]_{\mathbf{D}}$ from source language terms to semantic language terms. The two step nature is seen in the following diagram:



where for any **S**-program **s**

$$[[s]]_S = [[[[\mathbf{def}]]_{\mathbf{D}} s]]_{\mathbf{Sem}}$$

by definition. An equivalent statement: definition **def** is a compiler from **S** to **Sem**(!). Expressed symbolically we have

$$\mathbf{def} \in \begin{array}{c} \boxed{S \longrightarrow \mathbf{Sem}} \\ \boxed{D} \end{array}$$

7.3 Mosses' "Constructive Approach to Compiler Correctness"

The Morris/ADJ diagram can be simplified if we demand source and target meanings to be identical, and we do so in this and the next section. With this convention, Figure 7 illustrates Mosses' approach, where **T** is the target term algebra and $[-]_{\mathbf{T}}$ is its semantic function.

One of Mosses' ideas was to implement the semantic language once and for all by means of a derivor, and to *re-use* this fixed implementation to generate compilers for a variety of source languages. The essential property of the implementation can be described by:

$$\mathbf{imp} \in \begin{array}{c} \boxed{\mathbf{Sem} \longrightarrow \mathbf{T}} \\ \boxed{D} \end{array}$$

We said before that derivors can be composed symbolically, a fact we now state more formally using the "tee diagram" notation (proof is immediate):

⁷While λ -calculus is often used, it suffers a number of practical disadvantages which led Mosses to develop an alternative semantic framework called "action semantics". The choice of semantic language is not relevant to our discussion, so we say no more about this otherwise interesting subject.

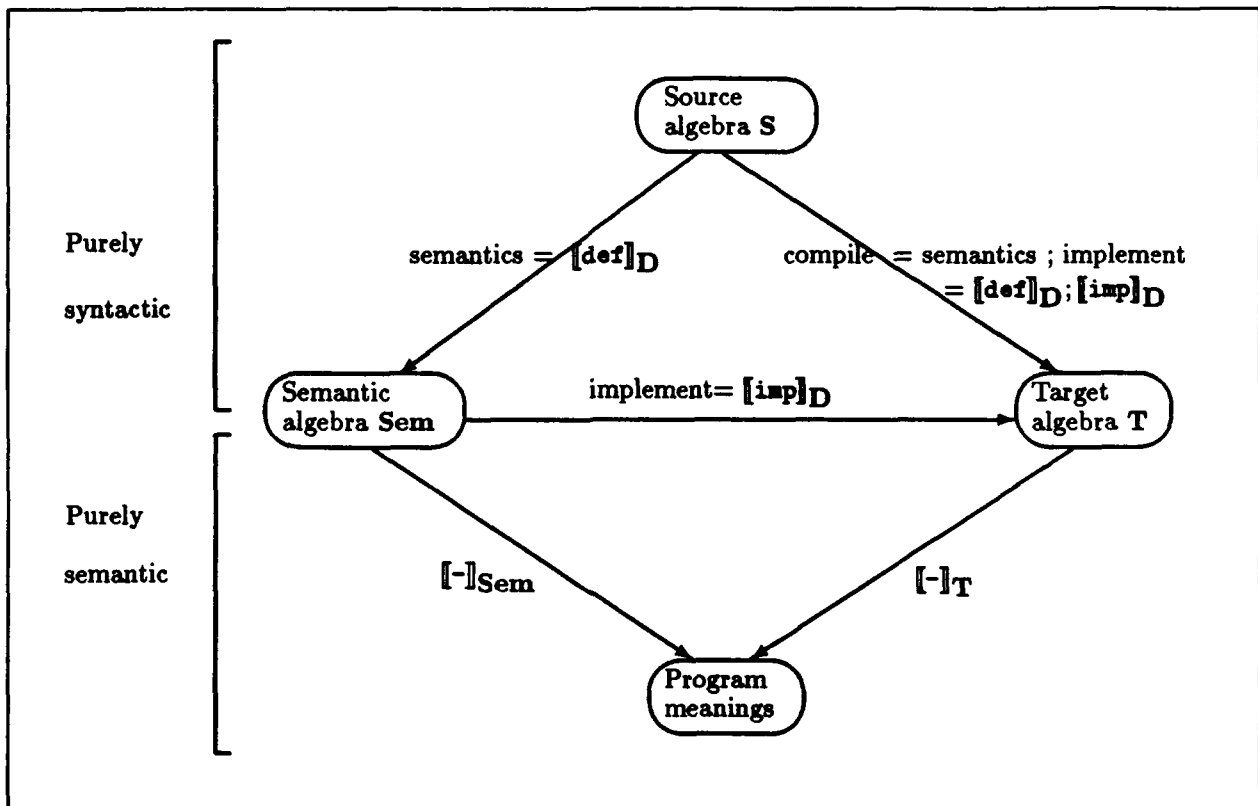


Figure 7: Mosses' Approach to Compiler Construction

Proposition 7.1 If $d \in \frac{A \rightarrow B}{D}$ and $p \in \frac{B \rightarrow C}{D}$ then

$$d ; p \in \frac{A \rightarrow C}{D}$$

A consequence is that compilers for various source languages may be obtained by symbolically composing their source semantics with the fixed implementation derivor **imp**. By the proposition just stated **comp** = **def** ; **imp** is a correct compiler from **S** to **T**, in the form of a derivor:

$$\text{comp} = \text{def} ; \text{imp} \in \frac{S \rightarrow T}{D}$$

Correctness of this compiler generation scheme is ensured by commutativity of the diagram. Note that it is only necessary to establish correctness of the lower triangle; then any source semantics whatever will be converted into a correct compiler by symbolic composition.

7.4 CERES

Mosses' approach is definitely a more general method for compiler generation than the Morris/ADJ technique. On the other hand, the constructed compiler is still a derivor, so compilation can only be done

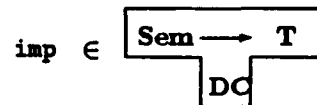
homomorphically. This limits the range of possible target programs, for instance identical subterms of a program will always be translated identically. Further, while derivors realize syntax-directed translations, they cannot express iterations that are unbounded according to source program structure, but such techniques are frequently seen in realistic compiler optimizations.

The starting point of the CERES approach to compiler generation [18] is to introduce a separate algebra C of *compile time actions*—a language appropriate for expressing compiler operations. As before the universal implementation imp is a derivor. However the diagram is changed so imp maps semantic expressions into “compile time actions” in C . The net effect is that compilation is not achieved by a derivor in one step, as with Mosses’ approach, but in two steps (conceptually at least).

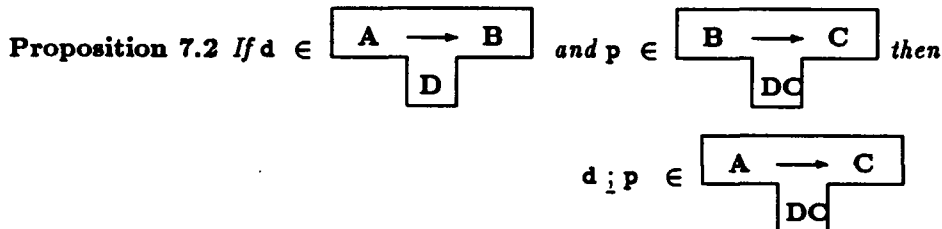
Given a Sem -term p , the first step is computation of $[\text{int}]_D p$. The result is a compile time action—a term in algebra C . The second step is evaluation of that compile time action via $[-]_C$. We invent a new language DC to describe this process, with definition:

$$[\text{def}]_{DC} p = [[\text{def}]_D p]_C$$

Given this, the type of imp can alternatively (and equivalently) be described by a tee diagram analogous to that of Mosses’ method:



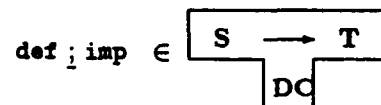
As by Mosses’ method, compiler construction is done by symbolically composing def with imp . The composition of def with imp is thus not a target program itself, but rather maps source S -programs to compile time actions which, when performed, construct the target program. A result analogous to the proposition above is easy to verify:



The net effect is to make possible more sophisticated compilation schemes than derivors alone can accomplish. Correctness of the scheme is expressed by commutativity of the diagram of Figure 8.

Producing Compilers in Target Code

The compilers resulting from this scheme are in language DC :



It is of course desirable to obtain a compiler in target language form:



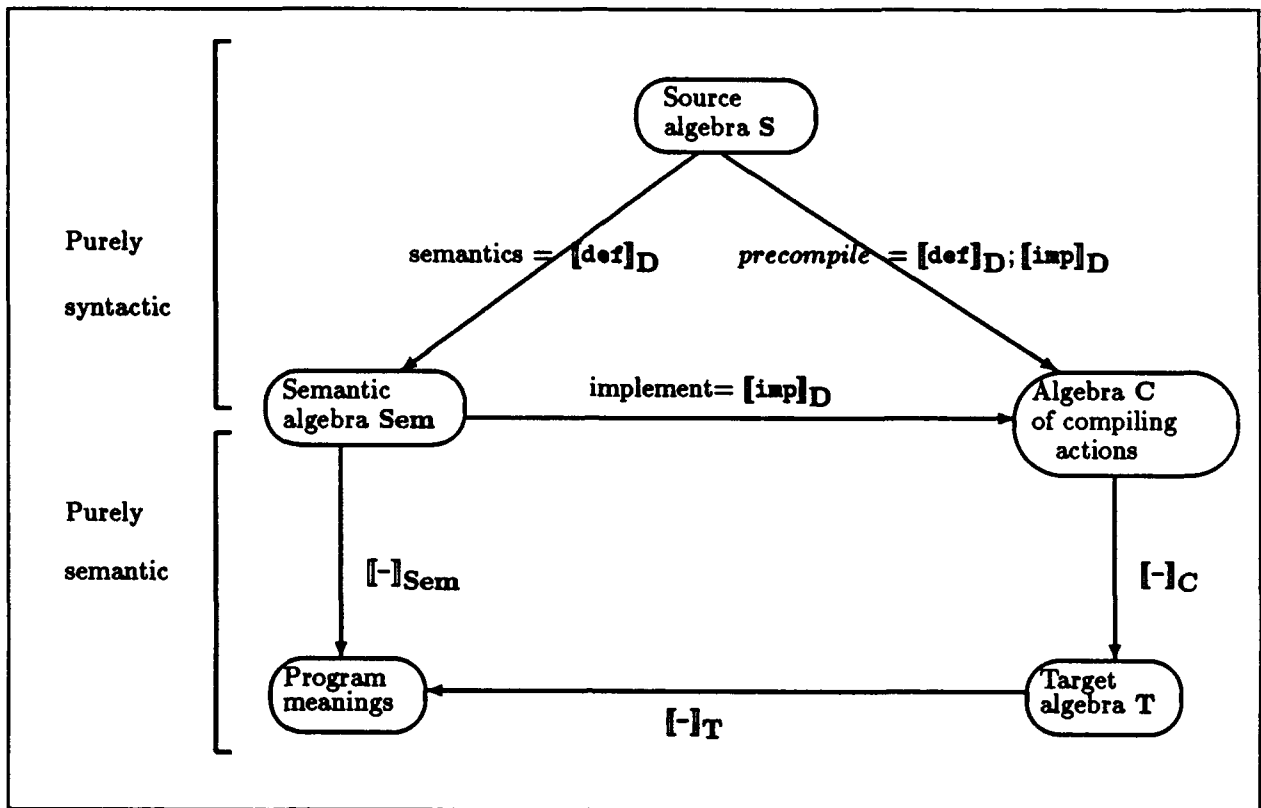


Figure 8: CERES Approach to Compiler Construction

This can be done with the aid of a bootstrapping process described in [18]. The technique is too involved to explain here, but is based on yet another example of symbolic computation called *self-composition* which we just mention in the next subsection. Surprisingly (and in spite of the two-step definition of language DC), the generated compilers have only one pass.

Yet Another Application of Symbolic Computation

The fundamental observation is that the essence of this approach to compiler generation is the transformation

$$\text{def} \Rightarrow \text{def} ; \text{imp}$$

This transformation will be performed for many user-written semantic definitions *def*, but all with the same implementation *imp*. Tofte shows in [18] that for any derivor *imp* there exists another derivor *imp'* with the property that for any derivor *def*,

$$[\text{imp}']_D \text{def} = \text{def} ; \text{imp}$$

In words, *imp'* accepts derivor *def* as its input data value, and yields as output the result of composing *def* with *imp*. A detailed explanation of how this leads to a practically useful technique for bootstrapping (as it does!) is beyond the scope of this article.

7.5 Problems for Study

1. Experiment with the CERES techniques using a stronger definition language, e.g. attribute-coupled grammars or a functional language.

2. Mosses' and CERES' techniques allow automation of code generation, but do not naturally handle compile-time computations, e.g. constant propagation, mapping variables to (base, offset) pairs, etc. Problem: extend the framework to allow a broader range of traditional compiling techniques.

8 Conclusions

Symbolic operations on programs have been seen to be widespread, and useful in several ways for compilation. A type system was developed that seems quite suitable for describing compilers, interpreters and other language processors. It was shown to give sensible types even in the case of the self-application used for generating compilers from interpreters via the Futamura projections. Further, the tee and box notations for compilers and interpreters have shown their descriptive powers.

A number of open problems have been stated along the way.

A Final Problem for Study is to develop a truly general framework for symbolic computation on programs—one in which symbolic composition, partial evaluation and other as-yet-unthought-of transformations can be expressed. A natural candidate would seem to be *Cartesian categorical combinators*, due their generality, high level (e.g. composition and currying are primitives), their proven utility in compiling (i.e. the Categorical Abstract Machine [4]), and the many algebraic laws that can be used to manipulate them.

References

- [1] Bjørner, D., A. P. Ershov, N. D. Jones, *Proc. Workshop on Partial Evaluation and Mixed Computation*, North-Holland, 1988
- [2] A. Bondorf, O. Danvy: Automatic autoprojection of recursive equations with global variables and abstract data types. Accepted by *Science of Computer Programming*, 1991.
- [3] A. Bondorf: Automatic autoprojection of higher order recursive equations. *ESOP Proceedings, Lecture Notes in Computer Science 432*, 1990, Springer-Verlag. Expanded version accepted by *Science of Computer Programming*, 1991.
- [4] G. Cousineau: The categorical abstract machine. Chapter 3, pp. 25-45, *University of Texas Year of Programming Series*, Addison-Wesley, 1990.
- [5] A. P. Ershov: Mixed Computation: Potential applications and problems for study. *Theoretical Computer Science* 18, pp. 41-67, 1982.
- [6] Y. Futamura: Partial Evaluation of Computation Process—an Approach to a Compiler-compiler. *Systems, Computers, Controls*, 2(5), pp. 45-50, 1971.
- [7] H. Ganzinger, R. Giegerich: Attribute-coupled grammars. *Proc. ACM Sigplan Symposium on Compiler Construction*, Montreal, Canada, 1984.
- [8] N. D. Jones, Peter Sestoft, Harald Søndergaard: MIX: A Self-applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, vol. 2, no. 1, pp. 9-50, 1989.
- [9] J. Goguen, J. Thatcher, E. Wagner: An initial Algebra approach to the specification, correctness and implementation of abstract data types. In *Current trends in programming Methodology IV* (ed. R. T. Yeh), pp. 80-149, Prentice-Hall, 1978
- [10] J. Goguen, J. Thatcher, E. Wagner: More on advice on structuring compilers and proving them correct. In *Semantics-Directed Compiler Generation*, pp. 165-188, Springer-Verlag 94, 1980.

- [11] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, T. Mogensen: A Self-applicable Partial Evaluator for the Lambda Calculus. IEEE Computer Society 1990 International Conference on Computer Languages, 1990.
- [12] Stephen Cole Kleene: Introduction to Metamathematics. North-Holland, 1952.
- [13] F. L. Morris: Advice on Structuring Compilers and proving them correct. 1st ACM Symposium on Principles of Programming Languages, pp. 144-152, 1973.
- [14] P. Mosses: SIS—Semantics Implementation System, Reference Manual and User Guide. DAIMI Report MD-30, University of Aarhus, Denmark, 1979.
- [15] L. Paulson: A Semantics-Directed Compiler Generator. 9th ACM Symposium on Principles of Programming Languages, pp. 224-233, 1982. Wolters-Noordhoff Publishing, 1970.
- [16] Hartley Rogers: Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.
- [17] D. Schmidt: Denotational Semantics: a Methodology for Language Development. Allyn and Bacon, 1986
- [18] Tofte, Mads, *Compiler generators - What they can do, what they might do and what they will probably never do*, 146 pp., EATCS Monographs, Springer-Verlag, 1990
- [19] V. F. Turchin: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems, 8(3), pp. 292-325, 1986.
- [20] V. F. Turchin: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems, 8(3), pp. 292-325, 1986.
- [21] P. Wadler: Listlessness Is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-time. ACM Symposium on LISP and Functional Programming, Austin, Texas, pp 45-52, 1984.
- [22] P. Wadler: Deforestation: Transforming Programs to Eliminate Trees. Proceedings of the Workshop on Partial Evaluation and Mixed Computation, North-Holland, 1988.

Contents

1	Introduction	1
1.1	High-level Operations in Programming Languages	1
1.2	Operations on Functions and Programs	2
1.3	Efficient Operations on Programs	2
1.4	Program Running Times	3
1.5	Goals of this Article	3
2	Definitions and Terminology	4
2.1	Programming Languages	5
2.2	Data and Program Types	5
2.3	Some Problems for Study	6
3	Efficient Symbolic Composition	7
3.1	Vector Spaces and Matrix Multiplication	7
3.2	Finite State Transducers	8
3.3	Derivors	8
3.4	Attribute Coupled Grammars	9
3.5	Functional programs	9
3.6	Some Problems for Study	9
4	Symbolic Function Specialization = Partial Evaluation	10
4.1	Function Specialization	10
4.2	Definition of Partial Evaluation	10
4.3	Some Techniques for Partial Evaluation	11
5	Compilers, Interpreters and Their Types	12
5.1	The Computational Overhead Caused by Interpretation	12
5.2	Can an Interpreter be Typed?	12
5.3	Well-typed Language Processors	13
5.4	Problems for Study	13
6	Partial Evaluation and Compiler Generation	13
6.1	The Futamura Projections	13
6.1.1	Compilation	14
6.1.2	Compiler Generation	14
6.1.3	Generating a Compiler Generator	14
6.2	Self-Application and Types	14
6.3	Efficiency Issues	16
6.3.1	Self-Application can Improve Efficiency	16
6.3.2	Optimality of Partial Evaluation	16
7	Symbolic Composition and Compiler Generation	17
7.1	The Morris and ADJ Advice on Compiler Construction	17
7.2	Denotational Semantics	17
7.3	Mosses' "Constructive Approach to Compiler Correctness"	18
7.4	CERES	19
7.5	Problems for Study	21
8	Conclusions	22

Uniform (Meta-) Development in the PROSPECTRA Methodology and System

Bernd Krieg-Brückner¹, Owen Traynor², Einar W. Karlsen³

In the PROSPECTRA Methodology and System, any kind of activity is conceptually and technically regarded as a transformation of a "program" in one of the system components. Programs are modelled as terms of an algebraically specified type; thus program and meta-program development use the same methodological basis. This provides for a uniform user interface, reduces system complexity, allows the construction of system components in a highly generative way, and is the basis for generalisation of specification, transformation, proof and development tactics, command language, even library access, and system configuration and development, into a single, unified framework.

1. Introduction

The project PROSPECTRA ("PROgram development by SPECification and TRANSformation") aims to provide a rigorous methodology for developing *correct* software and a comprehensive support system. It is a cooperative project between Universität Bremen (Prime Contractor), Universität Dortmund, Universität Passau, Universität des Saarlandes (all D), University of Strathclyde (GB), SYSECA Logiciel (F), Computer Resources International (DK), Alcatel Standard Electrica S.A. (E), and Universitat Politècnica de Catalunya (E) (subcontractor), sponsored by the Commission of the European Communities under the ESPRIT Programme, ref. #390 and #835.

The Methodology of Program Development by Transformation (based on the CIP approach of TU München integrates program construction and verification during the development process. User and implementor start with an algebraic requirement specification. This initial (loose) specification is then gradually transformed into an optimised machine-oriented executable program. The final version is obtained by stepwise application of transformation rules. These are applied by the system, with interactive guidance by the implementor, or automatically by compact transformation scripts. Transformations form the nucleus of an extendible knowledge base.

Overall, PROSPECTRA has not only achieved a powerful specification *and* transformation *language* with well-defined semantics that reflects the state-of-the-art in algebraic specification combined with higher-order functions, but also a comprehensive *methodology* covering the complete life-cycle (including re-development after revisions), integrating verification in a realistic way, supporting the development process as a computer-aided activity, and giving hope for a comprehensive formalisation of programming knowledge. Moreover, an integrated *prototype system* is operational, with a uniform user interface and library management including version and configuration control, that gives complete support and strict control of language and methodology to ensure correctness.

The paper first gives an introduction to the development and meta-development methodology, and then describes the effect of the uniform approach to the development of system components.

2. The Meta-Development Methodology

The methodology for program development based on the concept of algebraic specification of data types, and program transformation can be applied to the development of transformation algorithms, that is for program-manipulating programs or *meta*-programs. Starting from small elementary transformation rules that are proved correct against the semantics of the programming language, we can apply the usual equational and inductive reasoning to derive complex rules. Using all the results from program development based on algebraic specifications and functionals we can then reason about the development of meta-programs, i. e. transformation programs or development scripts, in the same way as about programs: we can define requirement specifications (development goals) and implement them by various design

¹Universität Bremen (D), ²University of Missouri St-Louis (USA), ³Computer Resources International (DK)

strategies; in short, we can develop *correct*, efficient, complex transformation operations from elementary rules stated as algebraic equations. Homomorphic extension functionals are important for the concise definition of program development tactics.

The meta-program development paradigm of PROSPECTRA leads naturally to a *formalisation of the software development process* itself. A program development is a sequence (more generally: a term) of transformations. The system automatically generates a transcript of a development "history"; it allows re-play upon re-development when requirements have changed, containing goals of the development, design decisions taken, and alternatives discarded but relevant for re-development.

In Program Development by Transformation, we can regard every elementary program development step as a transformation; we may conversely define a *development script* to be a composition of transformation operations (including application strategies for sets of elementary transformation operations). In this view we regard a development script as a *development transcript* (of some constant program term) to formalise a concrete development history, possibly to be re-played. A *development script* is, in general, a formal object that does not only represent a documentation of the past but is a plan for future developments. It can be used to abstract from a particular development to a class of similar developments, a *development method*, incorporating a certain strategy. Moreover, we can regard development scripts as formal objects: as (compositions of) such transformation operations. We can specify development goals, implement them using available operations, simplify development terms, re-play developments by interpretation.

The uniform approach to program, meta-program, proof and meta-proof development is perhaps the most important conceptual and methodological result of the PROSPECTRA project. But it also has had some major practical consequences. Since every manipulation in a program development system can be regarded as a transformation of some "program" (for example in the command language), the whole system interaction can be formalised this way and the approach leads to a uniform treatment of programming language, program manipulation and transformation language, proof and proof development language, and command language; the uniformity has also been exploited in the PROSPECTRA system yielding a significant reduction of parallel work.

The specification language PANndA-S (with Ada as a target) is also used as the transformation specification language TrafoLa-S. In this case, an abstract type schema to define Abstract Syntax is predefined, and translation to the applicative tree manipulation language of the Synthesizer Generator (used both as an Editor and as a Transformer Generator in the system, cf. [Reps, Teitelbaum 88a, b]) is automatic. It is quite important that abstract syntax trees are decorated with static semantic attributes, representing, for example, type information, but also the set of all applicable axioms (the "local theory") for efficiency of proofs and checking semantic applicability of transformations. ControLa, the command language of the system, is also a subset of TrafoLa: development histories can be simplified as formal objects, and replayed. Proof transformations are also written in TrafoLa-S and subsequently developed, in a transformational manner, into proof *tactics*.

3. The Generic Development System

The actual development system of for the PROSPECTRA methodology is a rather large and complex set of inter-related tools. The various components are brought together, in a unified way, by structuring the system to reflect the methodology it supports. The methodology itself has also been employed to develop and structure many of the constituent system components.

The PROSPECTRA system structure is notable for a number of reasons:

- (i) The orthogonality achieved by treating all systems activities as transformations
- (ii) The uniform interface to all system components
- (iii) The preservation of the 'simple' transformation paradigm even in the Meta Development Systems
- (iv) The uniform management of *all* system objects in the library (even parts of the system itself)
- (v) The description of the interface to all system components in the specification language PANndA-S

The relationship between the generic development system model and the (generic) transformational development model is rather straight-forward. We start the development with an initial requirements speci-

fication (using an *Editor* tool). The specification is then passed to a *Transformer* for further development and refinement. Correctness conditions are ensured by the *Proof System*. Finally, a *Target Generator* will generate an executable version of the objects produced by the Transformer and configure some environment for compilation/execution of these objects. All activities are recorded and carried out under the supervision of a controller in the context of some library. The *Library and Configuration Managers* coordinate and organise the various objects passed between and processed by the other system components.

Generation of a particular system component (or for a particular target language) requires some instantiation of the various generic components. For almost all kinds of developments, the required instantiation or specialisation of specific components can be done within the system; exceptions are, of course, compilers for the target languages into which the back ends translate.

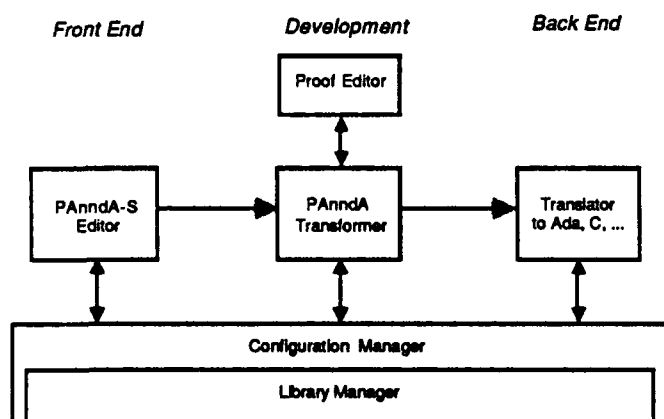


Figure 1: The Program Development Subsystem

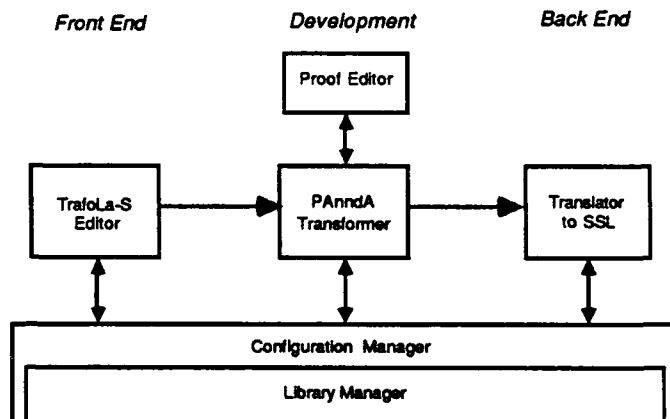


Figure 2: The Meta Program Development Subsystem

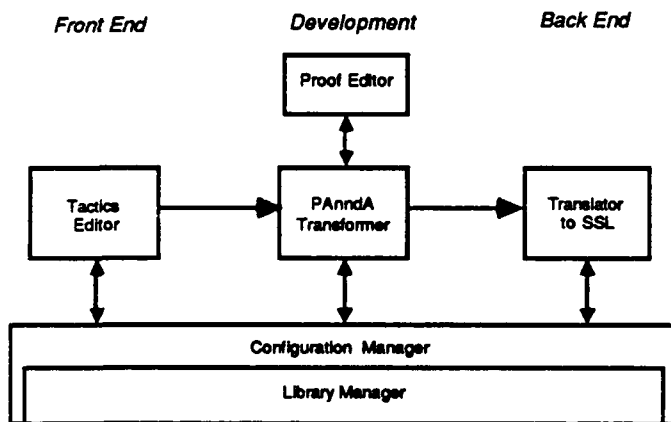


Figure 3: The Proof Tactics Development Subsystem

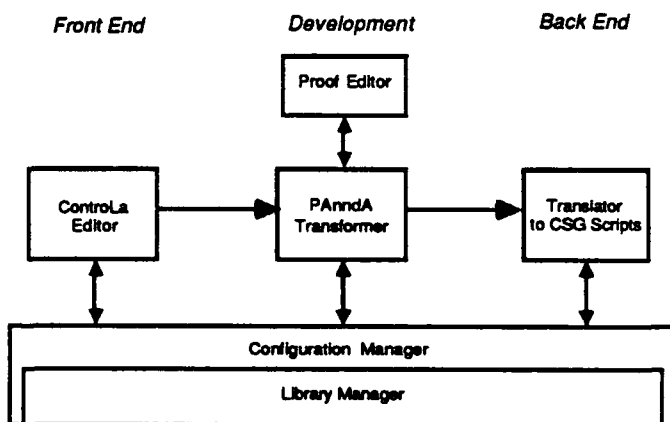


Figure 4: The System Development Subsystem

Activity

- Initial Formulation of Requirements
- Development to Constructive (and Optimised) Design
- Verification of some Correctness Conditions
- Translation to Target Language (Ada, C or other)

Component

- PAandA-S Editor
- PAandA Transformer
- Proof Editor
- Translator to Ada, C

Figure 5: Activities and Components for Program Development

As an example take the instantiation of the generic development system to the components used for program development in fig. 1. The table in fig. 5 illustrates the activities and the different components employed. This example shows the manner in which development proceeds in a typical subsystem. Each of the individual steps described varies in complexity. Some are completely automatic while others require a high degree of user guidance. In this particular case, the translator to Ada only unfolds abstract type definitions (introduced by pre-defined type definition schemata) into proper Ada text; in the case of C, for example, a little more work must be done.

4. Meta Development and System Development

Meta Program Development

Transformation development is done using the Meta Program Development Subsystem following the methodology for meta program development (cf. section 2). Transformation rules are specified in TrafoLa-S, the transformation specification sublanguage of PANndA-S. In addition to the PANndA-S constructs, it contains some syntactic abbreviations for phrases (PANndA program fragments in their concrete syntax form) and context-sensitive transformations to translate into canonical PANndA-S.

The subsystem for meta program development in fig. 2 is another example for an instantiation of the generic development system. The editor for TrafoLa-S is derived from that for PANndA-S; the requirement specification is frozen; a PANndA transformer with general transformations (and possibly special ones for meta development) is then applied to develop meta programs in the framework, and finally a particular kind of specification is compiled into an applicative tree manipulation program in SSL, the input language of the Synthesizer Generator. Note that in this case, objects produced by a particular target generator are subsequently incorporated into the system itself by the configuration manager. For example, a meta program for transformation may have been developed within the system; it is then compiled, integrated into a revision of the configuration of the transformer and may then be executed with that version of the transformer in subsequent sessions.

Proof Tactics Development

The proof system also has an associated meta development component. Proof Tactics Development (cf. fig. 3) is carried out in exactly the same way as meta program development. The resulting transformations for proofs are then analogous to the tactics used in the LCF system. An additional set of transformations is provided to allow the basic tactics for logical expressions to be developed into *true tactics*. Instead of merely transforming logical formulae, they generate proofs when applied to a logical formula (rather than just true or false). The proof trees correspond to the construction of a proof for a given logical formula.

System Development

The System Development Subsystem for the command language ControLa provides the facilities for enriching the development environment with new commands and scripts. Commands are formulated on the level of ControLa, an applicative subset of PANndA-S. ControLa specifications are entered using the ControLa editor, a restricted PANndA-S editor, which checks that the specifications are within the ControLa subset of PANndA-S. A target generator tool is provided for translating ControLa specifications into executable implementations in terms of CSG scripts. In addition, a translation is provided from abstract CSG scripts (the form in which concrete developments are recorded, cf. section 5.2) to basic ControLa expressions. Concrete developments can therefore be manipulated and abstracted on the level of ControLa using the PROSPECTRA methodology; this provides the basis for abstraction from particular developments to general methods that can be re-used. Thus the ControLa subsystem is both a System Development and a "Development Development" Subsystem.

5. References

- [Krieg-Brückner 88a] Krieg-Brückner, B.: Algebraic Formalisation of Program Development by Transformation. *in: Proc. European Symposium On Programming '88, LNCS 300* (1988) 34-48.
- [Krieg-Brückner 89a] Krieg-Brückner, B.: Algebraic Specification and Functionals for Transformational Program and Meta-Program Development. *in: Diaz, J., Orejas, F. (eds.): Proc. TAPSOFT '89 (Barcelona). LNCS 352* (1989) 36-59.
- [Krieg-Brückner 89b] Krieg-Brückner, B.: Algebraic Specification with Functionals in Program Development by Transformation. *in: Hünke, H. (ed.): Proc. ESPRIT Conf. '89, Kluwer Academic Publishers* (1989) 302-320.
- [Krieg-Brückner 90] Krieg-Brückner, B.: PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques Special Issue on Software Engineering in ESPRIT* (1990) 136-149.
- [Krieg-Brückner, Hoffmann 91] Krieg-Brückner, B., Hoffmann, B. (eds.): PROgram development by SPECification and TRAnsformation: Part I: Methodology, Part II: Language Family, Part III: System. PROSPECTRA Reports M.1.1.S3-R-55.2, -56.2, -57.2. Universität Bremen, 1990. (to appear in LNCS 1991).
- [Reps, Teitelbaum 88] Reps., Teitelbaum: *The Synthesizer Generator and The Synthesizer Generator; Reference Manual*. Springer, 1988.

Tools for algebraic distributed system design

Henk Eertink *
Tele-Informatics Group
University of Twente
eertink@cs.utwente.nl

1 Introduction

The design and construction of modern distributed systems is a complex job. High quality products can only be expected when some design methodology is adopted which provides both methods and tools to support the complete design trajectory. The ESPRIT project 2304 (LOTOSPHERE) aims at the development of such a methodology [6]. As this methodology is based on the use of a formal description technique, in this case LOTOS [2], it is possible to use tools to support this design process. Tools can be used for the specification of a system, the transformation of a specification into another, equivalent, specification and to generate (conformance) tests from a specification.

In this paper we will first give a short introduction to the LOTOS language and its formal model, then focus on LOTOS tools which support the design of distributed systems, and finally we will give a more detailed description of one of these tools, the simulator *smile*.

2 The specification language LOTOS

LOTOS (Language Of Temporal Ordering Specification) is a formal description technique, standardized by ISO. LOTOS integrates the specification of concurrent, communicating processes with the specification of abstract data types. The former is based on a process-algebraic calculus (much like CCS [5]), whereas the latter is based on many-sorted universal algebra.

A system as a whole is a single process, which may consist of several interacting subprocesses. A specification of a system is therefore essentially a hierarchy of process definitions. Each process can be seen as a black box, which is parameterized with a set of *gates*. These gates are the interaction points of that particular process on which it may communicate with its environment. Two or more processes may synchronize if they can all perform the same interaction (or *event*) at the same time. Upon synchronization, values may be interchanged between different processes. These values are represented using abstract data types. The specification of those ADTs is done in the ACT ONE language [1], which incorporates renaming, actualization, extension, parameterization and combination operators for the construction of data

*This work has been supported under the Esprit II program in project 2304 (LOTOSPHERE)

types. The internal structure of a process is defined by a so-called *behaviour expression*, in which the relation between the subprocesses is described. Primitive operators are the action prefix operator (which prefixes a behaviour expression with an event) and the process **stop** (inaction; it cannot perform any event). Other operators are the process instantiation, synchronization, interleaving, disable, sequential composition, hiding, local definition and choice operators.

The semantic model of LOTOS is a structured labelled transition system, consisting of a set of states and transitions between those states. A transition consists of a label (a gate name) and a finite string of data values taken from the union of the domains of the many-sorted algebra. (A transition is the formal interpretation of an event, whereas a state corresponds with a LOTOS behaviour expression).

3 Tools for LOTOS

The design of LOTOS specifications is supported by a large number of tools. Besides the classical language based tools like syntax checkers, cross reference report generators and static semantic checkers, the formal model of LOTOS allows the use of more sophisticated tools which really play a role in the design trajectory of a system. The design process of a system consists of different design steps, where in each step only a limited number of design decisions are considered and incorporated into the design. This design then acts as a starting point for a next step in which a more refined design is made. However, during this step, the user requirements must be preserved. The use of LOTOS allows us to use formal methods to *automate* such a design step. These methods can be supported by tools as well. Furthermore, the preservation of user requirements can be tested, by running test against a N-level specification. These tests can automatically be generated from the (N-1) specification, thus allowing greater confidence in the design decisions which are made during the design step.

To summarize, there are essentially three classes of LOTOS tools. The first class consists of tools which can be used to create a correct LOTOS specification. The prime example of such a tool is the LOTOS integrated editor [8]. This is a structure editor which incorporates a full static semantics checker and some report generation functions. Besides this tool, normal syntax and static semantic checkers are available as well. The dynamic behaviour of a LOTOS specification can be checked by a LOTOS simulator, which is discussed in the next section. The second class of tools consists of transformation tools. These tools can be used to transform a specification into another, equivalent, specification (which may be easier to implement). A special case of such a transformation tool is a compiler, which compiles LOTOS specifications into C-code [4]. The third class of LOTOS tools consists of test generation tools. These tools can be used to generate conformance tests for real implementations, or for checking whether a level (N+1) design conforms to a level N design.

A tool which falls in all three categories is a simulator. A simulator is used to explore the behaviour LOTOS specification, and as such can be used to gain more confidence in the specification ('It does what it is supposed to do'). But, as all possible events are generated, it is also quite easy to generate tests using such a tool. (The events which are possible now, should

be possible in subsequent designs as well). Some examples of such simulators are described in [3, 7].

If confluency is detected as well (i.e. the possibility to reach the same state through different paths), it is also possible to use a simulator to rewrite a specification with a lot of parallelism, possibly describing an infinite tree of behaviour, into a specification which contains no parallelism at all, thus allowing an easier implementation. The basic model of a simulator incorporating all these features is explained in the next section.

4 Simulation of formal specifications

In this section, the design of the symbolic LOTOS simulator *smile* will be discussed. *smile* is a tool for the exploration of the behaviour of full LOTOS specifications. It is fully symbolic, which means that the behaviour (which may be parameterized) can be studied without instantiating the parameters.

The tool consists of two different modules: one which expands behaviour expressions. The expansion function transforms a LOTOS behaviour expression B into an equivalent set of action prefixed behaviour expressions: $\text{expand}(B) = \Sigma\{g\bar{o}[P]; B' \mid B \xrightarrow{g\bar{o}[P]} B'\}$, where ';' stands for the action prefix operator and $\Sigma\{a, b, c\}$ means $a \parallel b \parallel c$ (\parallel is the LOTOS choice operator). '[P]' is the set of predicates on the event labelled with gate g where values \bar{o} are offered. These predicates are defined by the specification. For instance, if two processes synchronize on a gate g and the first process offers an arbitrary value x and the second process a value 4, a predicate $P = \{x = 4\}$ is generated. ' B ' is the resulting behaviour expression and can subsequently be expanded. The fact that B is capable of performing a certain event is denoted by $B \xrightarrow{g\bar{o}[P]} B'$ and is defined by the inference system of LOTOS.

This inference system consists of a number of inference rules (at least one for each LOTOS operator) and two inference axioms. The inference axioms are:

- (1) $\text{stop} \not\vdash$ (inaction; no events are possible!)
- (2) $g\bar{o}[P]; B \xrightarrow{g\bar{o}[P]} B$ (action prefix).

An example of an inference rule is the rule for the synchronization operator \parallel :

$$(3) \frac{B1 \xrightarrow{g\bar{o}1[P1]} B1' \quad B2 \xrightarrow{g\bar{o}2[P2]} B2'}{B1 \parallel B2 \xrightarrow{g\bar{o}1[P1+P2+P3]} B1' \parallel B2'}$$

where $P3 = \{o1_i = o2_i \mid i \leq |o1|\}$.

The synchronization is only possible if the sort of $o1_i$ is equal to the sort of $o2_i$, and if $|o1| = |o2|$.

The number of generated events is, in typical OSI applications, very large. To cut down this number, one must try to prove that the associated predicates P are equivalent with FALSE, regardless of the values of the free variables in the predicates. This task is performed by the second module in the tool: the lazy narrowing machine [9]. This module is developed at the Technical University of Berlin, and incorporates a so-called lazy narrowing algorithm. This algorithm can not only be used to find solutions for predicates, but also to prove that a

certain solution cannot be found. The use of this algorithm typically cuts down the number of events to about 10% of the original number.

An interesting aspect of *smile* is that the expansion function only *transforms* or *rewrites* the behaviour expressions into a semantically equivalent tree. In other words, the tool changes only the view, but not the meaning of the tree. This implies that the user always has a correct view of the specification, whatever he has expanded.

smile has been implemented completely in C and has an X-Windows based user interface. It runs approximately 4-8 times faster as his predecessor *hippo* [7]. It has been tested on Sun 3, Sun 4 (SPARC) and HP 9000 systems.

References

- [1] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, Berlin, 1985.
- [2] ISO. Information processing systems - open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour, IS 8807, 1989.
- [3] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software - Practice and Experience*, 18(4):365-385, April 1988.
- [4] J.A. Manas and T. de Miguel. From LOTOS to C. In K. J. Turner, editor, *Formal Description Techniques - Proceedings of the FORTE 88 Conference*, pages 79-84, Amsterdam, 1989. North-Holland.
- [5] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.
- [6] L. Ferreira Pires and C.A. Vissers. Overview of the Lotosphere Design Methodology. In *Proceedings Esprit Technical Week 1990, Brussels*, 1990.
- [7] Peter van Eijk. The design of a simulator tool. In P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 351-390. North-Holland, Amsterdam, 1989.
- [8] Peter van Eijk. LOTOS tools based on the cornell synthesizer generator. In H. Brinksma, G. Scollo, and C. A. Vissers, editors, *Proceedings of the ninth international symposium on protocol specification, testing and verification*, Amsterdam, 1989. North-Holland.
- [9] D. Wolz. Design of a compiler for lazy pattern driven narrowing. *Proc. of the 7th international workshop on specifications of abstract data types, Wusterhausen-Dosse*, Lecture Notes in Computer Science ???, 1990.

A Framework for Dexterous Manipulation Using Lie Algebras

Daniela Rus
Department of Computer Science
Cornell University

1 Dexterous manipulation

One of the great deficiencies of today's robots is flexibility. 85% of the robots used in industry are relegated to tasks like spot welding and spray painting. There are two main reasons why robots are limited today to such simple and repetitive tasks. First, their end effectors have a very simple, two stick structure. Second, dexterous manipulation, which is the manipulation of objects by anthropomorphic hands with independently moving fingers, is not well understood at present.

The effort to understand dexterous manipulation has, so far, been concentrated in the area of grasping. Work has also been done to study what a mechanical hand can do with an object. In [Li89], the author addresses the problems of grasp planning and motion planning for dexterous manipulation, by studying the motion of two rigid bodies under a rolling constraint. In [Bro87], the author provides a framework for studying object manipulation by multifinger hands by describing the set of possible infinitesimal motions of an object subject to a set of constraints.

We propose a new strategy for the autonomous manipulation of objects by multifinger robot hands, which we call *finger tracking* [Rus90]. Our notion of manipulation refers to the reorientation of an object by a mechanical hand by some degrees, about some axis, with respect to a fixed system of coordinates. The reorientation is accomplished by fine finger motions, and the hand never drops the object in the process.

2 Controlling dexterous manipulation

An observer of human manipulation would see clearly that a common strategy for object reorientation is to use fingers which are not involved in the grasp to move the object around. Finger tracking attempts to capture and formalize these ideas.

Our main assumption is that we have a polyhedral object, and a robot hand with enough fingers for a good grasp of the object, plus some extra fingers which we call *free fingers*. Some of the fingers are used to constrain the object, by restricting its degrees of freedom, and some of the others are used to generate its motion. Thus, given a polyhedron and a hand with n fingers, m of the fingers, $m \leq n - 1$, grasp the object, typically by assigning each finger to a separate face. The size of m depends on the contact type, as shown in [MNP90, MSS87, Ngu86]. Once the object is grasped, the m fingers stay fixed in space and the polyhedron is constrained to maintain continuous contact with them. Then a free finger pushes along some curve on a different face and causes the polyhedron to rotate relative to the grasping fingers. We call the action of the free finger *tracking*. The finger tracks a continuous trajectory, and at every point in time, the $m + 1$ fingers hold the object in a grasp with some desired property, for instance

equilibrium. This process allows the control of the axis and angle of rotation. The question we want to answer is how to track the free finger in order to generate some desired motion.

3 Using Lie algebras to specify control

A Lie group is a group which is also a manifold, and for which the composition and the inversion operations are smooth. The Lie algebra of a Lie group is the set of all left invariant vector fields on the group. Lie groups and Lie algebras form an area of mathematics in which modern algebra blends with classical analysis. Thus, they are a natural tool for the formalization of problems involving motion.

In order to make use of Lie algebras as computational tools for developing control algorithms for dexterous manipulation, we consider systems of the form

$$\mathcal{S} = (\mathcal{B}, \mathcal{F}, \mathcal{M}),$$

where \mathcal{B} is an object being manipulated by a hand \mathcal{F} in a configuration space \mathcal{M} . In this paper we are concerned with systems \mathcal{S} where \mathcal{B} represents a three dimensional body and contains the geometric information describing it. \mathcal{F} represents the hand, and contains information about the location of the contact points of the fingers onto the body, as well as the forces exerted by the fingers. \mathcal{M} represents the configuration space of motion for the body. Any motion of the object is a curve in this configuration space. The relationship between the components of \mathcal{S} is captured by two functions. The first function defines the configuration space associated with some given motion constraints, and it is denoted by

$$g : \mathcal{B} \rightarrow \mathcal{M}.$$

The study of g leads us to set of motions that could take place. The second function gives the connection between curves in configuration space and the forces exerted by the hand and it is denoted by

$$f : \mathcal{M} \rightarrow \mathcal{F}.$$

The study of f is essential in addressing various algorithmic questions about the control of dexterous manipulation. It could be used directly to decide what the hand should do in order to generate motion along a given curve in the configuration space. Its inverse could also be used to determine the resultant motion of the body, given a set of forces applied by the hand. These applications will be presented in a different paper. The rest of the section is devoted to deriving g and f .

Let P be a polyhedron, and let H be a robot hand with fingers f_0, f_1, f_2, f_3 . The vertices of P are denoted by p_i , and the finger contacts by q_i . P is constrained to keep contact with three of the fingers of H , which stay fixed in space. The other finger is a free finger, and it is used to generate the desired rotation of the object. The fingers have frictionless point contacts on P . Given P and H , \mathcal{B} and \mathcal{F} are easily derivable. We proceed by characterizing \mathcal{M} .

The motion of a body can be described as a curve in the group of rigid motions $SE(3)$ of the Euclidean space. Thus, \mathcal{M} is that subspace of $SE(3)$ which obeys some desired contact constraints. To be more precise, let T in $SE(3)$ be a matrix describing the position of the object relative to a frame of reference fixed on the robot hand, and $S = T^{-1}$ is a matrix describing the position of the robot hand relative to a frame of reference fixed on the object.

Then $S = \begin{pmatrix} R & u \\ 0 & 1 \end{pmatrix}$, where $R \in SO(3)$ is a rotation and u is a translation. The constraints that keep contact q_i on the face opposite vertex p_i are of the form $\det(q_i, T\bar{p}_j, T\bar{p}_k, T\bar{p}_l) = 0$. For simplicity of calculations, it is convenient to assume that p_l is the origin, and to express these constraints in terms of a reference frame fixed on the tetrahedron, as $\det(S\bar{q}_i, p_j, p_k, p_l) = 0$.

Theorem 3.1 *Given a polyhedron constrained to be in contact with three fixed points, the manifold of possible configurations of the polyhedron is given by*

$$\mathcal{M} = \left\{ \begin{pmatrix} R & u(R) \\ 0 & 1 \end{pmatrix} \mid u(R) = -\frac{\langle Rq_0, p_1 \times p_2 \rangle p_0}{\langle p_0, p_1 \times p_2 \rangle} - \frac{\langle Rq_1, p_2 \times p_0 \rangle p_1}{\langle p_1, p_2 \times p_0 \rangle} - \frac{\langle Rq_2, p_0 \times p_1 \rangle p_2}{\langle p_2, p_0 \times p_1 \rangle} \right\}$$

R is an element of $SO(3)$. The function relating translations to rotations captures $g : \mathcal{B} \rightarrow \mathcal{M}$.

Proof: With $\langle \cdot, \cdot \rangle$ denoting the dot product of two vectors, and $\cdot \times \cdot$ denoting the cross product of two vectors, the constraints simplify to $\langle Rq_i + u, p_j \times p_k \rangle = 0$; $\langle Rq_j + u, p_k \times p_i \rangle = 0$; $\langle Rq_k + u, p_i \times p_j \rangle = 0$. For any nondegenerate choice of vertices, p_i, p_j, p_k are linearly independent. This enables us to express $u = \alpha p_i + \beta p_j + \gamma p_k$. The coefficients α, β, γ are determined from the constraint equations. **QED**

Proposition 3.2 *The configuration space of motion, \mathcal{M} , is diffeomorphic to $SO(3)$.*

Each configuration of the polyhedron consistent with the given constraints is given by an element of \mathcal{M} . The motion of P is a continuous curve in \mathcal{M} . Since the groups $SE(3)$ and $SO(3)$ are inherent for \mathcal{M} , we want to make use of their structure and properties. $SE(3)$ and $SO(3)$ can be viewed as Lie groups. Thus, at each point in time, a snapshot of the configuration is represented by a Lie group element.

We continue this section with the study of f , the function relating the configuration space to the forces exerted by the fingers. f is determined by a system of equations consisting of the motions constraints, and the Newtonian force and torque equations. To derive this system, it is necessary to study the acceleration constraints associated with motions compatible with staying in \mathcal{M} . The structure of \mathcal{M} is complicated, and calculus on manifolds is hard. Thus, it is convenient to exploit the Lie group structure we have. For a Lie group, all tangent spaces may be naturally identified with the Lie algebra of the group, which is isomorphic to the tangent space at the identity of the group. Since we are interested in studying velocity and acceleration constraints, working with the tangent space at the identity is a tremendous simplification over the ordinary manifold case. The Lie algebra of $SE(3)$ has the representation

$$\mathcal{M} = \left\{ \begin{pmatrix} S & v \\ 0 & 0 \end{pmatrix} \mid S = -S^T \right\}$$

and the Lie algebra of $SO(3)$ is represented by the group of skew symmetric matrices, denoted by $SS(3)$. The velocity and acceleration constraints of \mathcal{M} can be obtained either by direct differentiation, or by moving to the Lie algebra of $SO(3)$. Calculations in $SS(3)$ are simpler than in $SO(3)$. For our particular application, if Y is the Lie algebra element corresponding to some rotation, the axis of rotation is defined by the three nonzero values of the matrix. The axis of rotation is less direct in $SO(3)$. We can show that if Q_t be a curve in $SO(3)$, then $Y_t = \dot{Q}_t Q_t^{-1}$ is a velocity curve in the Lie algebra of right invariant vector fields. By using this

result, we can show that the acceleration constraints are of the form $\det(\ddot{q}_{i_t}, p_j - p_l, p_k - p_l) = \det((\dot{Y}_t + Y_t^2)(q_{i_t} - u_t) + \ddot{u}_t, p_j - p_l, p_k - p_l) = 0$.

All the derivations done thus far are with respect to the object-fixed system of coordinates. This choice allowed us to use the face planarity of the polyhedron. This is not an inertial reference system, so we can not apply the usual force and torque equations. However, we could transform these equations so as to hold in the object-fixed frame, in terms of Lie algebra elements. Let M be the mass of P , r its center of mass, ν_i the face normals, I the inertia tensor, and Ω the angular velocity. Note that Ω is a vector defined by the nonzero values of a Lie algebra element Y . The force equation is:

$$(Y_t^2 - \dot{Y}_t)(r_0 - u_t) + 2Y_t \dot{u}_t - \ddot{u}_t = \frac{1}{M} \sum f_i \nu_i.$$

The torque equation is:

$$I\dot{\Omega} + M(\dot{u}_t \times (\Omega \times u_t) + u_t \times (\dot{\Omega} \times u_t) + u_t \times (\Omega \times \dot{u}_t)) = (I\Omega) \times \Omega + M(u_t \times (\Omega \times u_t)) \times \Omega + \sum_{i=0}^3 f_i q_i \times \nu_i - \sum_{i=0}^3 u_t \times f_i$$

By expressing all the constraint equations and the Newtonian force and torque equations in terms of Lie algebra elements of $SS(3)$, we derive the following result:

Theorem 3.3 *Given a polyhedral object and a robot hand with four fingers, three of which are fixed and one free to track, the essence of $f : \mathcal{M} \rightarrow \mathcal{F}$ is captured by a 4×4 linear system.*

4 Acknowledgements

I am pleased and grateful to acknowledge the many helpful discussions I have had with Allen Back and John Hopcroft during the course of this work. This research has been supported by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014-88K-0591, ONR Contract N00014-89J-1946, and NSF Grant IRI-9006137.

References

- [Bro87] D.Brock, *Enhancing the dexterity of a robot hand using controlled slip*, Technical Report MIT AI-TR-992, MIT AI Lab, 1987.
- [Li89] Z.Li, *Kinematics, Planning and Control of Dexterous Robot Hands*, PhD thesis, Berkley University, 1989.
- [MNP90] X. Markenscoff, L. Ni, C. Papadimitriou, *The geometry of form closure*, IJRR, Feb 1990.
- [MSS87] B.Mishra, J.T.Schwartz and M.Sharir, *On the existence and synthesis of multifinger positive grips*, Algorithmica, 2, 1987.
- [Nar88] S.Narasimhan, *Dexterous Robotic Hands: Kinematics and Control*, Technical Report MIT AI-TR-1056, MIT AI Lab, 1988.
- [Ngu86] V.Nguyen, *The synthesis of stable force-closure grasps*, Technical Report MIT AI-TR-905, MIT AI Lab, 1986.
- [Rus90] D.Rus, *Dexterous rotations of polygons*, Proceedings, Second Canadian Computational Geometry Conference, 1990.

Pattern Matching: A Sheaf-Theoretic Approach

Yellamraju V. Srinivas

Department of Information and Computer Science
University of California, Irvine, CA 92717, USA
srinivas@ics.uci.edu

We present a general theory of pattern matching by adopting an extensional, geometric view of patterns. The extension of the matching relation consists of the occurrences of all possible patterns in a particular target. The geometry of the pattern describes the structure of the pattern and the spatial relationships among parts of the pattern. The extension and the geometry, when combined, produce a structure called a sheaf. Sheaf theory is a well developed branch of mathematics which studies the global consequences of locally defined properties. For pattern matching, an occurrence of a pattern—a global property of the pattern—is obtained by gluing together occurrences of parts of the pattern—which are locally defined properties. Using sheaf theory, the geometric aspects of pattern matching can be treated algebraically.

Using such a characterization of pattern matching, we derive a generalized version of the Knuth-Morris-Pratt pattern matching algorithm [Knuth et al. 77] by gradually converting the extensional description of pattern matching as a sheaf into an intensional description.

1 The Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm [Knuth et al. 77] (hereafter abbreviated as “KMP”) is a fast pattern matching algorithm for finding occurrences of a constant pattern in a target string. It is linear in the sum of the sizes of the pattern and the target strings. KMP reduces the complexity of the naive algorithm for string matching (check for a match at every position in the target string) by avoiding comparisons whose results are already known (from previous comparisons). In particular, when a character mismatches after the pattern is partially matched, the next possible position in the target where the pattern can match can be computed by using the knowledge of the partial match. This “sliding” of the pattern on a mismatch is the most well known aspect of KMP. Here is an example, where there is a mismatch at the last character of the pattern, and the pattern can be slid three positions to the right.

slide	→	a b c a b a
pattern		a b c a b a
matches		✓ ✓ ✓ ✓ ×
target		a b c a b c a b c

The amounts by which to slide the pattern on possible mismatches can be precomputed in time proportional to the size of the pattern. Thus all occurrences can be enumerated in a single left-to-right scan of the target string without backing up.

The table assigning the amount of sliding to each mismatch is called the *failure function*. We attack the problem of rigorously deriving such a function from a specification of pattern matching. There are several derivations of KMP in the literature [Dijkstra 76, Bird et al. 89, van derWoude 89, Morris 90, Partsch and Völker 90]. However, all these derivations consider only pattern matching on strings. It is not apparent how to generalize these derivations because they crucially depend on the array representation of strings. We follow a more general approach of describing a match in terms of sub-matches; this description only depends on the geometry of the underlying data structures. We also explain the failure function as an instance of backtracking, a general strategy for searching.

2 A categorical/sheaf-theoretic description

We treat the data structures involved in matching (strings, trees, graphs, etc.) as objects in a category. The occurrence relation between patterns and targets is modeled as arrows. The geometry of the data structures (e.g., the decomposition of a graph into its edges) is described by a Grothendieck topology on the category.

DEFINITION 1: Sieve. A sieve S on an object a is a collection of arrows with codomain a which is closed under right composition, i.e., if $f: b \rightarrow a$ is in S , then for any arrow $g: c \rightarrow b$, the composite $f \circ g: c \rightarrow a$ is in S . A sieve S on an object a can be concisely represented as a sub-functor of the hom-functor $\text{hom}_{\mathcal{C}}(-, a)$. \square

DEFINITION 2: Grothendieck topology. A Grothendieck topology J on a category \mathcal{C} is an assignment to each object a of \mathcal{C} , a set $J(a)$ of sieves on a , called *covering sieves* (or just *covers*), satisfying the following axioms

1. (Identity cover)

For any object a , the maximal sieve $\{f \mid \text{codomain}(f) = a\}$ is in $J(a)$;

2. (Stability under change of base)

If $R \in J(a)$ and $b \xrightarrow{f} a$ is an arrow of \mathcal{C} , then the sieve $f^*(R) = \{c \xrightarrow{g} b \mid f \circ g \in R\}$ is in $J(b)$;

3. (Stability under refinement)

If $R \in J(a)$ and S is a sieve on a such that for each arrow $b \xrightarrow{f} a$ in R , we have $f^*(S) \in J(b)$, then $S \in J(a)$. \square

DEFINITION 3: Site. A site is a category along with a Grothendieck topology. The site formed by a topology J on a category \mathcal{C} will be denoted by (\mathcal{C}, J) . \square

For the pattern matching problem to be computationally tractable, and to simplify the derivation, we make some additional assumptions:

1. All objects in the site are finite.

2. A finest cover (i.e., a cover whose elements cannot be further decomposed) exists for each object.

3. All covers are strict epimorphic families, i.e., arrows defined on elements of a cover of an object determines a unique arrow on the object.

4. Every arrow in the site is an occurrence arrow.

Under these assumptions, the extension of the occurrence relation for a constant target and varying pattern, $\text{hom}_{\mathcal{C}}(-, t)$, forms a sheaf. A sheaf is a set-valued contravariant functor in which entities defined on elements of a cover of an object can be "glued" together to produce an entity defined on the object.

DEFINITION 4: Presheaf. A presheaf on a category \mathcal{C} is a contravariant functor from \mathcal{C} to the category of sets **Set**. \square

DEFINITION 5: Sheaf. A sheaf on a site (\mathcal{C}, J) is a presheaf $F: \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$ such that for every object a of \mathcal{C} and every covering sieve $R \in J(a)$, each morphism $R \rightarrow F$ in $\mathbf{Set}^{\mathcal{C}^{\text{op}}}$ has exactly one extension to a morphism $\text{hom}_{\mathcal{C}}(-, a) \rightarrow F$. \square

In the definition above, a morphism $\tau: R \rightarrow F$ is a natural transformation, and is a concise way of representing a *compatible family* of elements on a cover R . For a sieve $R = \{a_i \xrightarrow{f_i} a \mid i \in I\}$ on a , a compatible family of elements of F on the sieve R is a collection of elements $\{s_i \in F(a_i) \mid i \in I\}$, one for each arrow in the sieve R , which are compatible in the sense that for any arrow $u: a_i \rightarrow a_j$ in R for which $f_i = f_j \circ u$, the function $F(u)$ maps s_j onto s_i .

In figure 1, we show an example of the sheaf of occurrences of a pattern tree in a target tree. The underlying category consists of labeled trees as objects and label preserving maps as arrows. A cover of a

tree is a collection of subtrees such that for every edge in the original tree, there is at least one subtree in the cover which contains that edge. Every arrow is an occurrence arrow. In the figure, we show a specific target, a specific pattern, the finest cover for the pattern, and a part of the sheaf of occurrences. The subscripts are not part of the trees; they are a convenient notation for representing monic arrows. A sample compatible family of partial occurrences is shown with bold arrows. Dotted arrows indicate partial occurrences which do not give rise to any compatible families.

3 Derivation of a pattern matching algorithm

We follow the general heuristic of converting the extensional description of the occurrence relation as a sheaf into an intensional description (an algorithm). The sheaf condition immediately provides a divide-and-conquer strategy [Smith 85] for enumerating the occurrences of a pattern in a target:

1. (Decomposition step) Choose a cover for the pattern.
2. (Recursive invocation) Find occurrences of elements of the cover, i.e., find partial occurrences of the pattern.
3. (Composition step) Compose partial occurrences to obtain occurrences of the pattern.

We can eliminate the recursive invocation by using the stability of covers under refinement (the third axiom of a Grothendieck topology), and by choosing the finest cover for the decomposition. Finding occurrences of elements of the finest cover is usually trivial, e.g., enumerating identity arrows. Step 3 is the most complex part of the algorithm and consists of two parts:

- 3.1. Enumerate compatible families of partial occurrences on the chosen cover.
- 3.2. Glue together compatible partial occurrences to obtain total occurrences.

Gluing partial occurrences is usually simple and depends on the underlying data structures. Hence we concentrate on enumerating compatible families. Remembering that a compatible family assigns an occurrence arrow to each element of the pattern cover, we again use a divide-and-conquer strategy. Henceforth, we will call a compatible family a cone.¹ The decision to split the pattern cover into two pieces simplifies the composition step.

1. (Decomposition step) Split the pattern cover P into two pieces.
2. (Recursive invocation) Find all cones on each piece.
3. (Composition step) Combine the cones on pieces of the cover to obtain the cones on the pattern cover P .

To improve this algorithm, we make it incremental. We make a decision to sequentially traverse the target, rather than assuming that the extension of the sheaf is already given. Traversing the target inverts the computation of the cones; rather than actively assemble cones on pieces of the pattern cover, cones are assembled in the order given by the traversal. Such an incremental algorithm is obtained by applying finite differencing [Paige and Koenig 82] to the recursive cone assembly above.

1. Maintain a cache of partial occurrences; start with an empty cache.
2. Traverse the target, producing a stream of cones on pieces of the pattern cover.
3. For each increment, update all partial occurrences in the cache which are affected.

The incremental algorithm above is a naive algorithm which maintains all partial occurrences. To facilitate optimization, the problem is now reformulated as a search problem [Smith 88].

¹There is an isomorphism between compatible families represented as natural transformations and cones on the diagram corresponding to a sieve. See, for example, [SGA4, Exposé I, §3.5].

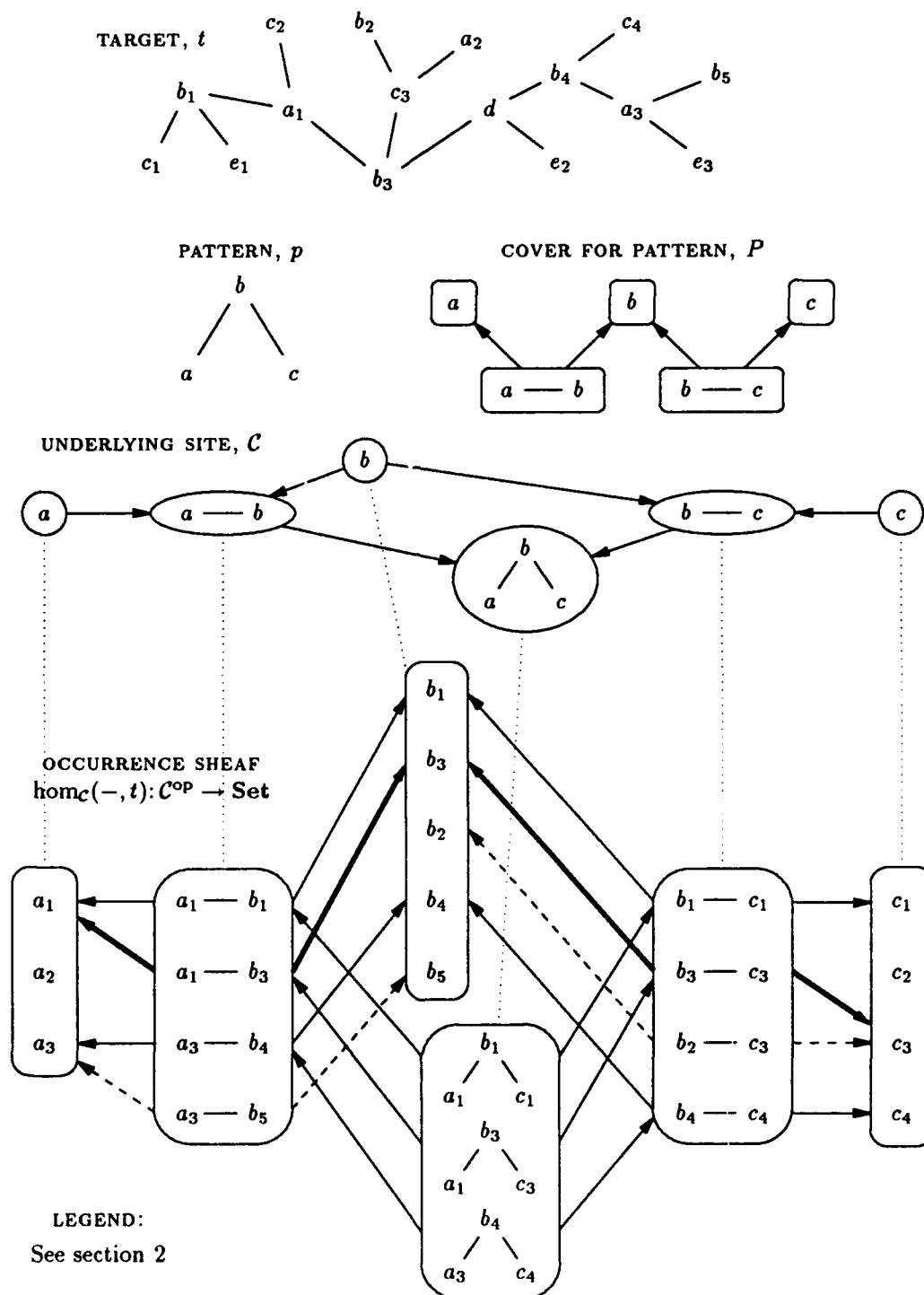


Figure 1: Sheaf of occurrences: example with trees

1. A state in the search space consists of a partial occurrence (a cone), and the unprocessed part of the stream of increments.
2. An operation in the state space consists of expanding the partial occurrence in a state using some increment in the stream.
3. The goal predicate tests for states which contain full occurrences.

The incremental algorithm above is equivalent to a breadth-first search of this state space. It can be improved by applying standard optimizations to the search. For example, we can prune away states which are not expandable. In the case of strings, with a left-to-right traversal of the target, we need only save those partial occurrences which touch the right end of the part of the target already traversed. This reduces the number of partial occurrences which have to be updated on each cycle of the incremental algorithm.

We can further reduce the number of partial occurrences at any stage by employing our overall heuristic of converting an extension into an intension: we replace a set of partial occurrences by a generator for that set. Such sets can be precomputed, as shown below.

Examination of the definition of compatible families reveals that, in a partial occurrence, each piece of target is associated with a piece of the pattern. In other words, each piece of the target is "parsed" as a piece of the pattern. Now, if a piece of the pattern occurs more than once in the pattern, then a piece of the target can generate multiple parses. The relationship of the pattern with itself can be precisely represented as the pattern-pattern-sheaf, i.e., the sheaf of occurrences of the pattern in the pattern.

By applying the first few steps of our derivation to the pattern-pattern-sheaf (observe that our derivation applies to any sheaf, not just to sheaves of occurrences), we can precompute the following:

for each partial occurrence ν , the set of partial occurrences which can be generated from ν by parsing its pieces differently.

Such alternative partial occurrences which are generated are said to be *subsumed* by the original partial occurrence. The subsumption relation forms a partial order. Using such precomputed sets, we can more efficiently search (breadth-first search with dependency-directed backtracking) the space of partial occurrences:

1. Try to expand the current set of partial occurrences using some parse of the current increment.
2. If some partial occurrence cannot be expanded, replace it with the largest partial occurrences subsumed by it and try again.

In the algorithm above, it may happen that for a particular increment, a partial occurrence has to be repeatedly replaced by subsumed partial occurrences until the increment can expand one. This repetition can be avoided by precomputing the appropriate subsumed partial occurrences for each partial occurrence and increment pair: this function is the failure function of the Knuth-Morris-Pratt algorithm!

4 A family of algorithms

The derivation above has a site (= category + topology) as a parameter. By appropriately instantiating the parameter, we get versions of KMP for strings, trees, graphs, etc.

Additionally, by choosing different alternatives in the design space, other algorithms can be derived. For example, in Waltz filtering [Waltz 75], the sheaf assigns a set of 3-D interpretations to each 2-D scene. The interpretations for commonly occurring junctions can be precomputed. The problem then is to choose a compatible set of junction labels to provide a consistent global interpretation. The difference in the derivation is where the precomputation is done.

The derivation can also be extended to multiple patterns and patterns with variables. An application of these extensions is the derivation of Earley's algorithm for context-free parsing algorithm. Context-free parsing can be treated as pattern matching with an infinite set of patterns. A context-free grammar acts as a generator for this infinite set. Earley's algorithm maintains a cache of partial parses. For each additional element of the target string, unexpandable parses in the cache are discarded; other parses are expanded by applying a grammar rule (thus generating new patterns to match). The difference in the derivation is that finest covers need not exist.

References

- [Bird et al. 89] BIRD, R. S., GIBBONS, J., AND JONES, G. Formal derivation of a pattern matching algorithm. *Sci. Comput. Programming* 12 (1989), 93-104.
- [Dijkstra 76] DIJKSTRA, E. W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [Knuth et al. 77] KNUTH, D. E., MORRIS, JR., J. H., AND PRATT, V. R. Fast pattern matching in strings. *SIAM J. Comput.* 6, 2 (June 1977), 323-350.
- [Morris 90] MORRIS, J. M. Programming by expression refinement: The KMP algorithm. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Springer-Verlag, New York, 1990, pp. 327-338.
- [Paige and Koenig 82] PAIGE, R., AND KOENIG, S. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 402-454.
- [Partsch and Völker 90] PARTSCH, H. A., AND VÖLKER, N. Another case study on reusability of transformational development: Pattern matching according to knuth, morris, and pratt. Tech. rep., KU Nijmegen, 1990.
- [SGA4] ARTIN, M., GROTHENDIECK, A., AND VERDIER, J. L. *Théorie des Topos et Cohomologie Etale des Schémas, Lecture Notes in Mathematics*, Vol. 269. Springer-Verlag, 1972. SGA4, Séminaire de Géométrie Algébrique du Bois-Marie, 1963-1964.
- [Smith 85] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27 (1985), 43-96.
- [Smith 88] SMITH, D. R. The structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, Palo Alto, California, July 1988. To appear in *Acta Informatica*.
- [van derWoude 89] VAN DER WOUDE, J. Playing with patterns, searching for strings. *Sci. Comput. Programming* 12 (1989), 177-190.
- [Waltz 75] WALTZ, D. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*. Mc-Graw Hill, New York, 1975, pp. 19-91.

Integration of semantical verification conditions in a specification language definition

*Didier Bert*¹
*Christine Lafontaine*²

Introduction

Programming in the large with specifications [EW86] requires an underlying language which verifies the correctness of the specifications operations [GH90], in the same way that a programming language provides type checking. This paper reports an attempt to find a formalism for defining algebraic specifications and operations on them in a secure way. To achieve this goal, verification conditions generated by the operations must be not only taken into consideration, but also integrated at the same level as the building primitives.

To identify the different difficulties, one uses the algebraic specifications as seen by LPG [BD90], i.e. based on an initial semantics for the Abstract Data Types. An important feature is the parameterization of the ADT by an equational theory [BE86]: the instantiation operation of generic specifications is formalized here. This paper shows that the Deva formalism [SW89] may serve to describe specifications, operations, verifications conditions, and proofs in a same framework.

The meta-level formalism

Deva is a generic system development language [SW89], partial synthesis of typed lambda-calculus, natural deduction and constructive logic. The theoretical approach stem from Nederpelt calculus [Ned80] and more recent work on the calculus of constructions [Coq87]. Deva relies on the principle of formulae-as-types and proofs-as-elements principle, i.e. a formula is identified with the type containing all its proofs. The language is higher-order in the sense that types obey the same rules as their elements. As a consequence, a single framework allows to express specifications, programs and developments.

¹LIFIA, Institut IMAG, av. Felix Viallet 46, F-38031 Grenoble Cedex (France).
e-mail: bert@uranus.imag.fr

²UCL, Unité d'Informatique, pl. Sainte Barbe 2, B-1348 Louvain-la-Neuve (Belgique).
e-mail: cl@info.ucl.ac.be

This research has been funded in part by the Belgian SPPS under grant RFO/AI/15

Since Deva allows their manipulation as objects, it is considered as a meta-language. This expressive power was already used to formalize VDM [Laf90] and the Bird-Mertens Calculus [Web90].

In the present LPG formalization, choosing Deva allows the manipulation of specifications as objects and the integration of validation conditions with their proofs. The interest is not to design a new specification language but to show that it is possible to express its semantics, and semantical conditions of correctness. Moreover, this research enforces a better understanding of the global context and a generalization of the concepts involved. In particular, the description of the LPG semantics [Rey87] in Deva led us to the definition of operations on specifications at a higher level than usual. A similar idea is proposed in [EM90] where the stress is on the construction of the specification rather than on the verification conditions.

Algebraic specifications and operations

A key concept of the paper is the notion of *effectiveness*, i.e. the ability to build the objects and to prove requirements about them. For algebraic specifications, basic objects are equational presentations [GB79]. The presentations considered here will be called *effective*: an e-presentation is made of a signature, and a formal system built over this signature, i.e a set of equations (axioms) and a set of inference rules.

A formula F is true in a e-presentation SP iff there exists a proof, i.e a text t obtained by composition of inference rules applied to axioms -in constructive logic, this means that the type of t is F . For equational presentations (i.e. the inference rules are those of the equality), this definition is equivalent to the membership of F to the theory presented by SP , where a theory is the class of formulae holding in the models of the presentation. For presentations of abstract data types, i.e. presentations with constraints, there is no longer equivalence between theories and theorems generated by inference rules [MS85]. Nevertheless, the *effective* approach is convenient because the aim is to be able to check all the conditions imposed by the specification language. Moreover, using Deva, specifications, operations on specifications, verification conditions and proofs are expressible in a single formalism. In other specification languages, like ASL [Wir86], the latter two aspects are not integrated in the language.

A specific and essential feature of LPG is the declaration and the use of specification morphisms. As usual, morphisms involve two specifications and a satisfaction condition. In Deva, a given element of type "morphism" is com-

posed of two e-presentations and the satisfaction proof. The integration of this proof is a real improvement with respect to other meta-languages. Some morphisms can be used to define generic data types in LPG. In this case, satisfaction is not needed because the e-presentation is copied into the generic one. However, like for a simple importation, one must require that the parameter (or imported) e-presentation is protected by the generic one. Here again, this protection condition can be expressed in Deva but, since this condition is undecidable, only *effective* protection is considered, i.e. protection which can be proved. To achieve this goal, a theory of the protection has to be defined, based on sufficient conditions to prove this property.

Several operations on e-presentations are defined in this framework. We present two of them here. The first one is the composition of morphisms of equational e-presentations. This operation takes two morphisms $A \rightarrow B$ and $B \rightarrow C$ as parameters and returns the morphism $A \rightarrow C$. This operation must compute the proof that the equations of A hold in C , given that the equations of A hold in B and these of B in C . It is clear that this definition is only possible with a meta-language dealing with proofs as objects. The second operation which has been expressed in Deva is the instantiation of a generic e-presentation. Like in the parameter passing technique [EK84], the arguments of the instantiation are three e-presentations: the parameter P , the generic specification G and the actual parameter M , together with the two morphisms $\Phi_r: P \rightarrow G$ and $\Phi_m: P \rightarrow M$. As expected the result is the e-presentation G instantiated by M (GM), and the morphism $\Phi_i: G \rightarrow GM$.

Conclusion

This work consisted in defining the basic concepts of algebraic specifications. The language Deva requires clear and complete notations, leading to a formalization which is complex but structured and free from tacit assumptions. With the notion of e-presentation, an attempt is done to internalize the external constraints. The description of operations on the e-presentations involve syntactical and semantical aspects. The treatment of verification conditions and proofs play a central rôle to enforce correctness of specification programming.

Bibliography

- BD90** Bert D., Drabik P., Echahed R., Declerfayt O., Demeuse B., Shobbens P.-Y., and Wautier F., *Reference Manual of the Specification Language LPG*, RT-LIFIA, 1990.
- BE86** Bert D., and Echahed R., Design and Implementation of a Generic, Logic, and Functional Programming Language. In: Robinet B., and Wilhelm R. (eds), *Proc. ESOP'86, European Symposium on Programming*, LNCS 213, pp.119-132, 1986.
- Coq87** Coquand Th., *Une théorie des Constructions*, PhD thesis, Univ. Paris VII, 1987.
- EK84** Ehrig H., Kreowski H.-J., Thatcher J., Wagner E., and Wright J., Parameter Passing in Algebraic Specification Languages. In: *TCS 28*, pp.45-81, North-Holland, 1984.
- EM90** Ehrig H., and Mahr B., *Fundamentals of Algebraic Specifications 2: Modules Specifications and Constraints*, Springer-Verlag, 1990.
- EW86** Ehrig H., and Weber H., *Towards Programming in the Large with Algebraic Module Specifications*, IFIP Congress, Dublin, 1986.
- GB79** Goguen J.A., and Burstall R.M., The Semantics of CLEAR, a Specification Language. In: Bjorner D. (ed.), *Abstract Software Specification*, Proc. Copenhagen Winter School, LNCS 86, pp.292-332, 1980.
- GH90** Guttag J.V., and Horning J.J., Formal Specifications as a Design Tool. In: *POPL 7th*, pp.251-261, 1980.
- Laf90** Lafontaine Ch., Formalization of the VDM reification in the DEVA meta-calculus. In: Broy M. and Jones C.B. (eds), *Proc. of the IFIP TC2 Working Conference on Programming Concepts and Methods*, 1990.
- MS85** MacQueen D.B., and Sannella D.T., Completeness of Proof Systems for Equational Specifications. In: *IEEE Trans. Software Engineering*, SE-11(5), pp.545-560, 1985.
- Ned80** Nederpelt R.P., An Approach to Theorem proving on the Basis of a typed Lambda-calculus. In: Springer verlag (ed.), *Proc. of the fifth conference on Automated Deduction (LNCS 87)*, pp.182-194, 1980.

- SW89** Sintzoff M., Weber M., de Groote Ph., and Cazin. J, *Definition 1.1 of the generic Development language DEVA*, RR89-35, Université Catholique de Louvain, Unité d'Informatique, 1989.
- Rey87** Reynaud J.-C., *Sémantique de LPG*, RR-651, IMAG 56, LIFIA, March 1987.
- Web90** Weber M., Formalization of the Bird-Mertens algorithmic calculus in the DEVA meta-calculus. In: Broy M. and Jones C.B. (eds), *Proc. of the IFIP TC2 Working Conference on Programming Concepts and Methods*, 1990.
- Wir86** Wirsing M., Structured Algebraic Specifications: a Kernel Language. In: *TCS 42*, pp.123-249, 1986.

An Algebraic Semantics for the Specification Language Z^{++}

K. Lano, H. Haughton

April 1, 1991

Abstract

This paper describes a formal semantics for the object-orientated specification language Z^{++} , [9], and gives applications of this semantics to the definition of refinements and transformations of specifications in the language. The language itself is an extension of Z [12], and uses algebraic specifications of class behaviour as external specifications of these classes, which may in addition have internal state-based specifications of their behaviour. The algebraic description is used to satisfy proof obligations required of systems which use this class, and the state-based description is used in refinement.

1 An Overview of Z^{++}

Object-orientated concepts have become widely recognised as useful in making both development and maintenance of software more efficient and effective. Z^{++} extends Z by providing an explicit notation for defining classes of objects, and was devised to improve the maintainability and ease of refinement of large systems. It has been used as a representation language for reverse-engineered data-processing systems in the REDO project [8]. Examples illustrating the advantages of the encapsulation and separation of concerns provided by the object class style of description are given in [9, 10]. Here we will detail the syntax and the low-level semantics of object classes.

1.1 Class Syntax

The BNF description of a Z^{++} class declaration is:

```
Object_Class ::= CLASS Identifier TypeParameters
                [EXTENDS Imported]
                [TYPES Types]
                [FUNCTIONS Ardefs]
                [OWNS Locals]
                [INVARIANT Predicate]
                [RETURNS Otypes]
                [OPERATIONS Otypes]
                [ACTIONS Acts]
                END CLASS

TypeParameters ::= [ Idlist ]
                  |  $\epsilon$ 

Imported       ::= Idlist

Types          ::= Type_Declarations

Locals         ::= Identifier : Type ; Locals
                  | Identifier : Type

Otypes         ::= Identifier : Idlist  $\rightarrow$  Idlist ; Otypes
                  | Identifier : Idlist  $\rightarrow$  Idlist

Acts           ::= [Expression &] Identifier Idlist ==> Code ; Acts
                  | [Expression &] Identifier Idlist ==> Code
```

The *TypeParameters* are a list (possibly empty) of generic type parameters used in the class definition. The *Types* are type declarations of type identifiers used in declarations of the local variables of the object. The *Local* variable declarations are variable declarations. The **OPERATIONS** list declares the types of the operations, as functions from a sequence of input domains to an output domain. The **RETURNS** list of operations defines the output type of those attributes and functions of the objects internal state that are externally visible; these are operations with no side-effect on the state, and it has been found helpful in practice [15] to distinguish these from operations that do change the state. The **INVARIANT** gives a predicate that specifies the properties of the internal state, in terms of the local variables of the object. This predicate is guaranteed to be true of the state of an object class instance between executions of the operations of the object instance.

The **ACTIONS** list the definitions of the various operations that can be performed on instances of the object; for instance we would write:

$$READ\ x \implies q' = tail\ q \wedge x = head\ q$$

in a specification of queues with contents q .

The input parameters are listed before the output parameters in the action definitions. *Code* includes Z predicates and procedural UNIFORM [13] code; both have a precise semantics [3].

2 The Institution Semantics of Z^{++}

We will use a modified version of the concept of an institution [5], to provide a semantics for classes.

Definition The *theory* Γ_C of a class C is defined to be the first-order extension of ZF which has constants \underline{c} for each attribute name c of C , together with those constants inherited from each attribute type that is itself a class, type symbols \underline{T} for each attribute or parameter type of C , function symbols \underline{f} for each method f of C , and for each method of an inherited class, and axioms:

- (1): $\underline{c} \in \underline{T}$ for each attribute declaration $c : T$
- (2): Inv_C in terms of the \underline{c}
- (3): Each axiom of Γ_D , for each class D used as an attribute type in C
- (4): $\forall x : \underline{X}; v : \underline{T}(Inv(v) \wedge (y, v') = Op(x, v) \Rightarrow \theta(x, v, y, v') \wedge y \in \underline{Y} \wedge v' \in \underline{T} \wedge Inv(v'))$

where in (4), Op is a method $Op : X \times T \rightarrow \mathbf{P}(Y \times T)$ of C , with definition

$$Op\ x\ y \implies \theta(x, v, y, v')$$

Models of the theory Γ_C correspond to instances of the class C .

Definition A *refinement* $\phi : C \rightarrow D$ between two object classes, written as $C \sqsubseteq D$, is defined to be a theory morphism: $\phi : \Gamma_C \rightarrow \Gamma_D$, which is a map that provides a type symbol $\phi(M)$ of Γ_D for every type symbol M of Γ_C , a constant symbol $\phi(d)$ for every constant symbol d of Γ_C , and a function symbol $\phi(Op)$ of Γ_D for every function symbol Op of Γ_C , such that $\Gamma_D \vdash \phi(\Gamma_C)$. This is the same as requiring that D can provide a relative interpretation of C . We allow definitional extension of Γ_D by means of function composition, and allow functions to be built out of old ones by means of taking a cross product with an identity function.

There is a category of (theories of) object classes, in which refinement is the categorical arrow, and in which generic classes are representable as endo-functors, ie, instantiation preserves refinement. The category has products and co-products, $\Gamma_C + \Gamma_D$ is the least refined class that inherits both D and C , whilst $\Gamma_C \times \Gamma_D$ is the conjunction, the greatest common abstraction, of two classes. The empty class (with no attributes or methods, and with invariant **true**) is initial in the category, and any class with **false** invariant and at least one method and attribute is terminal in the category.

2.1 Characteristic Algebra

We can also extract an algebra from the theory of a class. Define the *term algebra* Σ_C of a class C to be the set of formal terms (1): x where x is a member of the term algebra of the attribute or parameter

types of C , and (2): $op((x_1, \dots, x_n), a)$ where op is a method of C , and has input and output parameters $in_1, \dots, in_n, out_1, \dots, out_p$, and a stands for the object instance that is changed by application of op . x_1, \dots, x_n and a are terms of Σ_C . The result is a pair $((y_1, \dots, y_p), a')$ standing for the result of the method, together with the changed class. When $p = 0$ only the changed class is represented as a result of the method. Initialisation operators correspond to constant members of the class. Thus, in a class which implements a stack, we have formal terms *Initial*, *Push*($\underline{2}$, *Initial*), and so forth. Then the algebra A_C of the class is the quotient of the set of ground terms of form (1) or (2) of Σ_C by the equalities between terms provable under the theory Γ_C .

If $C \sqsubseteq D$ via ϕ , then A_C is a subalgebra of A_D via the embedding

$$\begin{aligned} h_\phi : A_C &\longrightarrow A_D \\ t &\longmapsto [\phi(t)] \end{aligned}$$

where $[\phi(t)]$ is the equivalence class of $\phi(t)$ under the equalities introduced by the theory of D .

By the definition of refinement $\Gamma_C \vdash t = t'$ implies that $\Gamma_D \vdash \phi(t) = \phi(t')$, so this map is well-defined. It will be injective only if Γ_D does not introduce new equalities between terms of C : this means it is *hierarchically consistent* in the sense of [6], or, in the case of inheritance, that D creates no confusion in the sense of [5]. It will be surjective only if every term in A_D can be expressed as the image of a term of A_C under the map h_ϕ : that is, the refinement is *relatively complete*, or introduces 'no junk'.

Algebraic requirements on a class C , in the form of universally quantified equations on the terms of Γ_C , create a quotient of Γ_C by these new equations, and a corresponding refinement of the class. However this refinement may be to the terminal class.

3 Application of the Semantics to Refinement

A large library of class transformation rules are available [11], which preserve or refine the meaning of a class with respect to the above semantics. For instance it is easily proved that the addition of a new attribute, or the strengthening of a class invariant, are both refinement operations. Using these rules for development gives a different approach to that usually applied to Z, since this transformational process considers a specification as a single entity, rather than as a universally visible set of operations and states, without modularisation.

Work on strategies for the direct refinement of algebraic specifications to procedural code [4] extends [14], and allows us to bypass the explicit state-based specifications of classes in certain cases.

4 Expressing the semantics of Z^{++} within Z^{++}

We can express the semantics of our language within itself [11], via the 'class of classes', which specifies how object instances are generated, deleted and made the subject of a method application. This allows us to define more powerful operations, such as a method that applies a method to each of the currently existing instances of a class, and to give a semantics for these operations.

5 The Relationship with FOOPS

Transformations from FOOPS [7] into Z^{++} , and from a subset of Z^{++} to FOOPS can be defined, which enable us to use the semantics for FOOPS to give an alternative semantics for parts of Z^{++} .

6 Conclusion

We have defined a semantics for a wide-spectrum objected-orientated language, using the institution of classical predicate calculus. This logical basis is more general than the equational logics used by Goguen to give a semantics for FOOPS [5], however a link with executability is retained, since we can compile these theories into the language of a theorem-proving tool such as the B tool [1], and then use these to investigate the properties of the specification. Animation tools for object classes have also been developed.

References

- [1] *The B User Manual*, Draft 1.4, January 1991, BP Ltd, Sudbury, UK.
- [2] J. Bowen, R. Gimson, S. Topp-Jorgensen, *Specifying System Implementations in Z*, Oxford University Programming Research Group Technical Monograph PRG-63, 1988.
- [3] P. Breuer, K. Lano, *From Code To Z Specifications*, Proceedings of the 4th Z User Meeting, Z User Workshop, Springer-Verlag Workshops in Computing, J. Nicholls (Ed), September 1990.
- [4] H. Haughton, *A Refinement System for Algebraic Specifications*, Lloyds Register of Shipping, Croydon, 1991, to be submitted to the BCS Journal of Formal Aspects of Computing.
- [5] J. A. Goguen, *An Algebraic Approach To Refinement*, VDM '90, VDM and Z, Lecture Notes in Computer Science 428, Springer-Verlag 1990.
- [6] J. A. Goguen, T. W. Thatcher, E. G. Wagner, *An Initial Approach to the Specification, Correctness and Implementation of Abstract Data Types*, Current Trends in Programming Methodology, Number 4, Chapter 5, 1978.
- [7] J. E. Goguen, J. Meseguer, *Unifying Functional, Object-Orientated and Relational Programming with Logical Semantics*, SRI International 1987.
- [8] T. Katsoulakis, *The REDO Project: Maintenance, Validation and Reverse-Engineering*, ESPRIT 1989 Conference, Brussels.
- [9] K. Lano, *Z⁺⁺, an Object-Orientated Extension to Z*, Proceedings of the Fifth Annual Z User Meeting, Oxford University, December 1990. To appear in Springer-Verlag Workshops in Computer Science.
- [10] K. Lano, *Integrating Development and Maintenance in an Object-Orientated Environment*, REDO Project Document 2487-TN-PRG-1050, Oxford University PRG.
- [11] K. Lano, *Expressing Z⁺⁺ in Z⁺⁺*, REDO Project Document 2487-TN-PRG-1056, Oxford University PRG.
- [12] M. Spivey, *The Z Notation : A Reference Manual*, Prentice Hall 1989.
- [13] C. Stanley-Smith, A. Cahill, *UNIFORM: A Language Geared To System Description and Transformation*, University of Limerick, 1989.
- [14] M. Thomas, *The Imperative Implementation of Algebraic Data Types*, Dept. of Computer Science Technical Report TR44, January 1988, Stirling University.
- [15] J. A. Zimmer, *Restructuring For Style*, Software - Practice and Experience, Vol. 20(4), 365 - 389.

ABOUT ALGEBRAS, FIXPOINTS AND SEMANTICS

Irène GUESSARIAN *

C.N.R.S. - L.I.T.P. Université Paris 6

4, Place Jussieu, 75252 Paris Cédex 05, France

e-mail: ig@litp.ibp.fr

I INTRODUCTION

It now is well known that the methods of universal algebra can be applied to programming languages, thus yielding an Algebraic Semantics [Andréka-Németi, ADJ, Guessarian]. Many problems in semantics can be expressed within the framework of universal algebra, which provides partial answers for them. In turn, this often leads to new questions, henceforth new problems in universal algebra, thus resulting in a two way communication channel between the two disciplines. We will, in the present paper, illustrate this situation with two examples: the first one consisting in an application of algebraic semantics to logics of programs, and the second one in an application of algebraic semantics to a fixpoint semantics of true concurrency.

The main goal of algebraic semantics has been to provide a clean and sound semantics of programming languages by splitting the syntactic and semantic parts of a program as much as possible. Using tools of universal algebra, one can then describe abstractly, i.e. independently of any interpretation, the syntactic properties of a program ; after which, one is well equipped to translate these abstract or syntactic properties, via any concrete interpretation, into concrete or semantic properties of the real program. Moreover, this can be done stepwise, introducing at each step exactly the needed amount of semantic knowledge. Thus algebraic semantics makes easier and more natural the concepts of modularization and abstraction that are essential in software development.

In algebraic semantics, interpretations are nothing but certain algebraic systems in the sense of [Mal'cev], or equivalently, Σ -structures or models in the sense of [Grätzer]. The main tool for characterizing the syntax of a program is to have it compute symbolically in a free interpretation, i.e. an absolutely free algebra of terms; the set of results of all possible symbolic computations is then represented by an infinite tree, i.e. an infinite term, that characterizes the behaviour of the program with respect to all interpretations. Properties of programs are thus described by properties of the associated infinite tree. One of the main problems one has to deal with, in proving any program property, is to prove equivalences of programs; this approach shows that two programs are equivalent with respect to all possible interpretations if and only if their associated infinite trees are equal.

In practice however, equivalence with respect to all possible interpretations will be too exacting. We have to take into account some of the constraints or properties verified by the interpretations we are interested in; this extra information should even be modularized according to need. We thus can find alternate syntactic objects and structures to describe the set of all possible symbolic computations under some given constraints. The constraints are described by a class of interpretations \mathcal{C} , and there exists, for any given class \mathcal{C} , a generic - or free, or Herbrand - interpretation H , together with ways of describing:

- (1) the free interpretation H
- (2) the equivalence with respect to H ; this equivalence will in turn characterize the equivalence with respect to the class \mathcal{C} .

This H corresponds roughly to what is usually called a free \mathcal{C} -algebra, or an algebra that is free (initial) relative to the class \mathcal{C} of algebraic systems [Guessarian]. The description of the free interpretation H , its elements, and its equivalence can then be applied fruitfully to prove various kinds of program properties.

In algebraic semantics, one can study several interesting types of classes of interpretations: (in) equational classes (defined by a set of (in)equations), algebraic classes (where, intuitively, any (in)equation between

* Support from the PRC Mathématiques-Informatique is gratefully acknowledged.

programs can be proved by computation induction, or properties of finite subterms extend by continuity into properties of infinite terms), and first-order definable classes. In the present paper, we will study two applications of (in)equational classes:

- the first one will be an application to logics of programs: we will show how to deduce from the free interpretation a complete proof system for deriving all valid (in)equations with respect to some class \mathcal{C} .
- the second one will be an application to the semantics of concurrency: we will show how to obtain a semantic model for traces of concurrent processes as a factor algebra of an Herbrand interpretation H . This second example will also illustrate the idea of modularization: we will first construct a model for non deterministic processes, then express true concurrency with the help of the non deterministic model (and interleaving), which is the technique used in the algebras of processes for CCS or ACP ([Milner, Bergstra-Klop]).

The paper is organised as follows: section 2 contains the preliminaries and notations for algebraic semantics; section 3 shows how to deduce a proof system complete for (in)equational logics, from the Herbrand model; finally, section 4 describes how to obtain the model of true concurrency we are interested in, starting from the Herbrand interpretation H , and using fixpoints and morphisms.

II ALGEBRAIC SEMANTICS

II.1 Algebras

Let Σ (resp. Φ) be a finite ranked alphabet of base function symbols (resp. of variable function symbols). The rank of a symbol s in $\Sigma \cap \Phi$ is denoted by $r(s)$. Symbols in Σ are denoted f, g, h, \dots if they have rank ≥ 1 , and a, b, \dots if they have rank 0; Σ_i denotes the symbols of rank i in Σ . Symbols in Φ are denoted ϕ, ψ, \dots . Let X be a set of variables: the variables have rank 0 and are denoted by u, v, w, \dots possibly with indices.

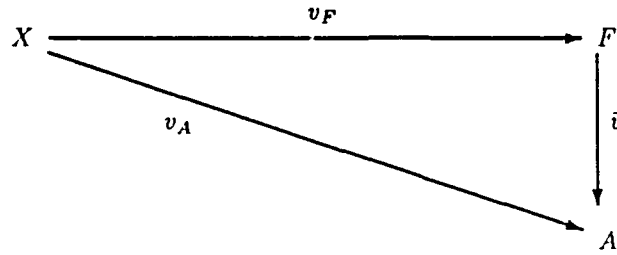
Let T_Σ denote the set of trees, or terms, well-formed over Σ (see [Guessarian] for basic notions about trees and operations on trees). A tree with variables in X is a tree over $\Sigma \cup X$, i.e. an element of $T_{\Sigma \cup X}$: intuitively, the variables are intended to range over a set of trees, e.g. T_Σ or $T_{\Sigma \cup X}$. T_Σ (resp. $T_{\Sigma \cup X}$) is endowed with a Σ -algebra structure: for f in Σ , f of rank p , and t_1, \dots, t_p in T_Σ (resp. $T_{\Sigma \cup X}$), the operation f_{T_Σ} (resp. $f_{T_{\Sigma \cup X}}$) is defined by: $(t_1, \dots, t_p) \mapsto f(t_1, \dots, t_p)$. T_Σ (resp. $T_{\Sigma \cup X}$) is the domain, or carrier set, of the free initial Σ -algebra, or Σ -magma in the terminology of [Nivat 75], (resp. the free Σ -algebra over generators X , $T_{\Sigma \cup X}$, which is also denoted by $T_\Sigma(X)$). This will be justified by the fact that T_Σ and $T_{\Sigma \cup X}$ have the free property, which we will recall below in full generality. We will in the sequel identify T_Σ (resp. $T_{\Sigma \cup X}$) with the corresponding free algebra.

Let us first recall the notions of ordered and complete Σ -algebras, which we will use later. A Σ -algebra A , or algebraic structure of similarity type Σ , consists of a carrier set A and for each function symbol σ in Σ , a function $\sigma_A : A^w \rightarrow A$ where $A^w = A \times \dots \times A$ with $w = \text{rank}(\sigma)$, and σ_A is a constant in A if $w = 0$.

An ordered Σ -algebra is a Σ -algebra such that the carrier set A is endowed with an ordering \leq_A and a least element \perp_A , and the operations σ_A are order preserving. Ordered Σ -algebras are actually a special case of algebraic structures of similarity type $\Sigma' = \Sigma \cup \{\leq\}$, where $\Sigma_\perp = \Sigma \cup \{\perp\}$. The class of all ordered Σ -algebras forms a quasivariety, i.e. is axiomatisable by quasi-atomic formulae, in the sense of [Mal'cev, Grätzer]. This will be made more explicit through an example in the next section.

A Σ -algebra A is said to be *complete* iff, all directed subsets of A have a lub (least upper bound) in A , and the σ_A 's are continuous, i.e. preserve lub's of directed sets. The category of Σ -algebras (resp. ordered Σ -algebras, or complete Σ -algebras) is defined as follows: objects are Σ -algebras (resp. ordered Σ -algebras, or complete Σ -algebras); morphisms are Σ -homomorphisms (resp. order-preserving Σ -homomorphisms, i.e. Σ' -homomorphisms, or continuous Σ -homomorphisms that is: homomorphisms which, in addition to preserving the Σ -structure, also preserve lub's of directed sets, which implies that they preserve \perp and the order also). Recall that a Σ -homomorphism from a Σ -algebra A to a Σ -algebra B is a function $f : A \rightarrow B$ that satisfies: for σ in Σ with $\text{rank}(\sigma) = n$, and $a_i \in A$ for $i = 1, \dots, n$: $f(\sigma_A(a_1, \dots, a_n)) = \sigma_B(f(a_1), \dots, f(a_n))$; and for σ in Σ_0 : $f(\sigma_A) = \sigma_B$. Note finally that a Σ -homomorphism $h : A \rightarrow B$ is said to be *strict* if and only if: $h(\perp) = \perp$.

A (complete, ordered) $\Sigma \cup X$ -algebra F is said to be *free over generators* X iff for any (complete, ordered) $\Sigma \cup X$ -algebra A there exists a unique Σ -homomorphism (in the category of complete or ordered Σ -algebras) $\bar{h} : F \rightarrow A$ making the following diagram commute:



where, for any $\Sigma \cup X$ -algebra J , $v_J(x) = x_J$, for all x in X . F is also called the free (complete, ordered) Σ -algebra generated by X .

The free Σ -algebra generated by X , also called the algebra of Σ -terms with variables in X , and denoted by $T_\Sigma(X)$, exists and can be constructed as follows: its carrier set is the set of well-formed terms (or trees) on the alphabet $\Sigma \cup X$, and is defined inductively by:

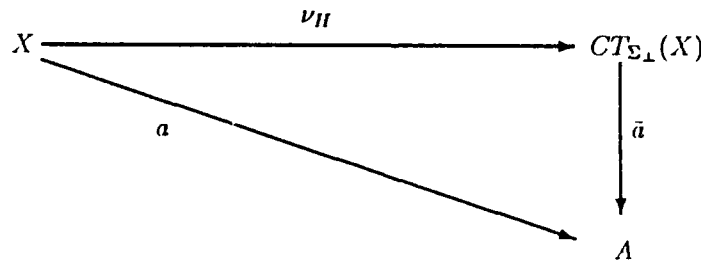
- (i) $\Sigma_0 \cup X \subseteq T_\Sigma(X)$,
- (ii) $\sigma(t_1, \dots, t_n) \in T_\Sigma(X)$ for each σ in Σ and t_i in $T_\Sigma(X)$ for $i = 1, \dots, n$, and $\text{rank}(\sigma) = n$.

The Σ -algebra structure of $F = T_\Sigma(X)$ is defined by: $\sigma_F = \sigma$ if $\sigma \in \Sigma_0$, and otherwise: $\sigma_F(t_1, \dots, t_n) = \sigma(t_1, \dots, t_n)$, for t_i in $T_\Sigma(X)$ for $i = 1, \dots, n$.

The free ordered Σ -algebra $F' = T_{\Sigma_\perp}(X)$ generated by X has carrier sets the sets of well-formed terms on the alphabet $\Sigma \cup X \cup \{\perp\}$; its Σ -algebra structure is defined as the one of F ; its carrier set F' is endowed with the least ordering such that: (i) \perp is the least element, and (ii) the Σ -algebra operations are order-preserving.

Finally, the free complete Σ -algebra $H = CT_{\Sigma_\perp}(X)$ generated by X is the ideal completion of F' (see [Birkhoff]). $H = CT_{\Sigma_\perp}(X)$ is the set of ideals of F' , ordered by inclusion, and the Σ -algebra operations are extended by continuity; namely for σ in Σ and I_i ideal of F' for $i = 1, \dots, n$, $\sigma_H(I_1, \dots, I_n)$ is the ideal generated by $\{\sigma_F(i_1, \dots, i_n) / i_j \in I_j \text{ for } j = 1, \dots, n\}$. H can be viewed as the set of well-formed finite and infinite trees generated by X . Its ordering \leq extends the ordering on F' and can be intuitively described by: $T \leq T'$ iff T' can be deduced from T by substituting some occurrences of \perp by terms different from \perp .

By the freeness of $H = CT_{\Sigma_\perp}(X)$, we know that for each valuation $a : X \rightarrow A$ in an arbitrary complete Σ -algebra A , there exists a unique morphism \bar{a} making the following diagram commute:



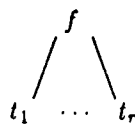
For T in $CT_{\Sigma_\perp}(X)$, having variables $\{x_1, \dots, x_n\}$ we will denote by T_A the application $A^n \rightarrow A$ defined by $a = (a_1, \dots, a_n) \mapsto \bar{a}(T)$, i.e. every n -tuple $a \in A^n$ determines a valuation $X \rightarrow A$, and we define $T_A(a) = \bar{a}(T)$.

Let CT_Σ (resp. $CT_\Sigma(X)$) be the subset of $CT_{\Sigma_\perp}(X)$ consisting of terms without occurrences of \perp . T_Σ (resp. CT_Σ) is the set of finite (resp. finite and infinite) trees on the alphabet Σ ; similarly, $CT_\Sigma(X)$ is the set of finite and infinite trees on $F \cup X$. Among these trees, some are of particular interest for semantics.

Throughout this paper we will use the word notation for trees. A tree is represented by a word over the alphabet $\Sigma \cup \{(\cdot), \cdot, c\}$, where c is the comma, as follows:

a stands for $a \in \Sigma_0$

$f(t_1, \dots, t_r)$ stands for the tree having root $f \in \Sigma_r$ and direct subtrees t_1, \dots, t_r and would be pictured as:



A tree t in $T_{\Sigma, p}(X)$ will be denoted by $t(x_1, \dots, x_p)$ in order to point out the variables; the shorthand vector notation $t(\vec{x})$ will be used: $t(t_1, \dots, t_p)$ denotes the tree obtained by simultaneously substituting t_i for each occurrence of x_i in $t(x_1, \dots, x_p)$, for $i = 1, \dots, p$; formally, $t(t_1, \dots, t_p)$ is the image of $t(x_1, \dots, x_p)$ by the

$\Sigma \cup X$ -algebra morphism h defined by: $h(x_i) = t_i$ for $i = 1, \dots, p$, and $h(f) = f$ for f in Σ . We will also use a vector shorthand notation $t(\vec{t})$ for $t(t_1, \dots, t_p)$. Alternatively, when wanting to point out which trees are being substituted for which variables, we will note: $t(t_1/x_1, \dots, t_p/x_p)$ (or $t(\vec{t}/\vec{x})$).

II.2 Recursive schemes, rational and algebraic trees

DEFINITION II.1 A recursive scheme on Σ is a pair (S, t) , where S is a system of n equations:

$$S: \quad \phi_i(x_1^i, \dots, x_n^i) = t_i \quad , \quad i = 1, \dots, n \quad (1)$$

where, for $i = 1, \dots, n$, $\phi_i \in \Phi$, $x_j^i \in X$ for each j , $t_i \in T_{\Sigma \cup \Phi}(X)$, and $t \in T_{\Sigma \cup \Phi}(X)$.

Each scheme is associated with a term rewriting system (or context free tree grammar) which is denoted by S_{\perp} , and defined by :

$$\phi_i(x_1^i, \dots, x_n^i) \rightarrow t_i + \perp$$

Let $t' \Rightarrow_S t''$ denote the rewriting relation associated with S_{\perp} , i.e. t'' is deduced from t' by substituting t_i or \perp for one occurrence of some ϕ_i in t' . Let $\stackrel{*}{\Rightarrow}_S$ denote the reflexive and transitive closure of \Rightarrow_S , and $L(S, t) = \{t'/t' \in T_{\Sigma \cup \Phi}(X) \mid t \stackrel{*}{\Rightarrow}_S t'\}$ be the tree-language generated by S with axiom t (cf. [Guessarian]). It is well-known that $L(S, t)$ is a directed subset of $CT_{\Sigma \cup \Phi}(X)$, let $T(S, t) = \text{lub} L(S, t)$ in $CT_{\Sigma \cup \Phi}(X)$. Note that there are no longer occurrences of \perp in $T(S, t)$. Formally, immediate rewritings \Rightarrow_S and derivations $\stackrel{*}{\Rightarrow}_S$ according to S are defined as usual. Let us recall the definitions here for the reader's convenience. The immediate rewriting according to S is the relation \Rightarrow_S defined on $T_{\Sigma \cup \Phi} \times T_{\Sigma \cup \Phi}$ by: $t \Rightarrow_S t'$ iff there exists some prefix \underline{t} of t , an occurrence o in \underline{t} labeled by the function variable ϕ_i of rank n (i.e. $t(o) = \phi_i$), and subtrees t_1, \dots, t_n of t such that: $t = \underline{t}(\phi_i(t_1, \dots, t_n)/o)$ and $t' = \underline{t}(t_i(t_1, \dots, t_n)/o)$, or $t' = \underline{t}(\perp/o)$.

Intuitively, at occurrence o , ϕ_i is macroexpanded, i.e. replaced by the right handside t_i of production $\phi_i(x_1, \dots, x_n) \rightarrow t_i$; simultaneously, t_1, \dots, t_n are substituted for x_1, \dots, x_n . Whenever no ambiguity can occur, the subscript S is omitted and \Rightarrow_S (resp. $\stackrel{*}{\Rightarrow}_S$) is denoted by \Rightarrow (resp. $\stackrel{*}{\Rightarrow}$), as will be done in the sequel.

A recursive scheme is said to be *iterative* iff all function variables in Φ are of rank 0, or, from a programming viewpoint, iff all the t_i 's are left-linear: intuitively, an iterative scheme corresponds to left-linear or terminal recursion, which is well-known to be equivalent to iteration. Left linear terms of $T_{\Sigma}(X)$ are defined inductively by:

- terms in $T_{\Sigma} \cup \Phi_0$ are left-linear
- for f in Σ_n , and t_1, \dots, t_n left linear terms, $f(t_1, \dots, t_n)$ is left-linear.

In a left-linear term t , function variables from Φ can label only the leaves of t , since $\Phi = \Phi_0$, the set of function variables of rank 0.

The solution of a recursive (resp. iterative) scheme is called an *algebraic* (resp. *rational*, or *regular*) tree. The terminology comes from language theory, because algebraic trees are obtained as languages generated by context-free tree grammars, and rational trees are generated by regular tree grammars.

Rational (resp. algebraic) trees are trees which are solutions of iterative (resp. recursive) schemes, and correspond intuitively to unfoldings of iterative, i.e. WHILE-loop programs (resp. LISP-like recursive programs).

EXAMPLE II.2 A recursive scheme and its algebraic tree

Having defined a binary operation *mult* on a data type, we want to extend that operation to an operation *lmult* on list of elements of that data type by means of the recursive LISP like program P:

$$\begin{aligned} \text{mult}(L, L') &= \text{if } L = \text{NIL then NIL else} \\ &\quad \text{if } L' = \text{NIL then NIL else} \\ &\quad \text{cons}(\text{mult}(\text{car } L, \text{car } L'), \text{lmult}(\text{cdr } L, \text{cdr } L')) \end{aligned}$$

where *car* L (resp. *cdr* L) is the first element (resp. the rest) of the list L .

To this recursive program, corresponds a program scheme, naturally obtained by abstracting the meanings of the functions if-then-else, car, etc..., and replacing as much as possible of the right hand side of the above equation by a single function name. We obtain here the scheme S , where $\Phi = \{\text{lmult}\}$, $X = \{L, L'\}$, $\Sigma = \{\perp, \text{cdr}, g\}$.

$$S: \quad \text{lmult}(L, L') = g(L, L', \text{lmult}(\text{cdr } L, \text{cdr } L'))$$

The solution of scheme S (and the corresponding program) is then computed by successive approximations. The n th approximation σ_n to that solution is defined by:

1) unwinding first the scheme S n times, i.e. replacing n times all occurrences of $lmult$ by the right-hand side of S , starting with the term $lmult(L, L')$. This gives here, after n unwindings $s_n(L, L') = g(L, L', g(cdr L, cdr L', g(\dots, g(cdr^{n-1} L, cdr^{n-1} L', lmult(cdr^n L, cdr^n L')) \dots)))$.

2) replacing then all occurrences of $lmult$ in s_n by \perp (the totally undefined function) yielding here: $\sigma_n(L, L') = g(L, L', g(cdr L, cdr L', g(\dots, g(cdr^{n-1} L, cdr^{n-1} L', \perp) \dots)))$.

σ_n belongs to $L(S, lmult(L, L'))$, but not s_n . In the present case, evaluating σ_n yields:

$$\begin{aligned} \sigma_n(a_1 a_2 \dots a_p, a'_1 a'_2 \dots a'_{p'}) &= mult(a_1, a'_1) mult(a_2, a'_2) \dots mult(a_q, a'_q) \\ &\quad \text{if } q = \inf(p, p') < n \\ &= \perp \quad (\text{the undefined element) otherwise.} \end{aligned}$$

We choose here the simpler but ambiguous notation of christening of the same name \perp the undefined elements, or equivalently least elements, of the various domains, syntactic or semantic (data, lists of data), that we consider.

Thus the σ_n make their domain of definition grow larger and larger. That results in a chain $\sigma_0 < \sigma_1 < \dots < \sigma_n < \sigma_{n+1} < \dots$ where the ordering $<$ represents the relation: "to be less defined than".

A tree like representation of S , the s_n 's and σ_n 's which might help the intuition is given in Figure 2.

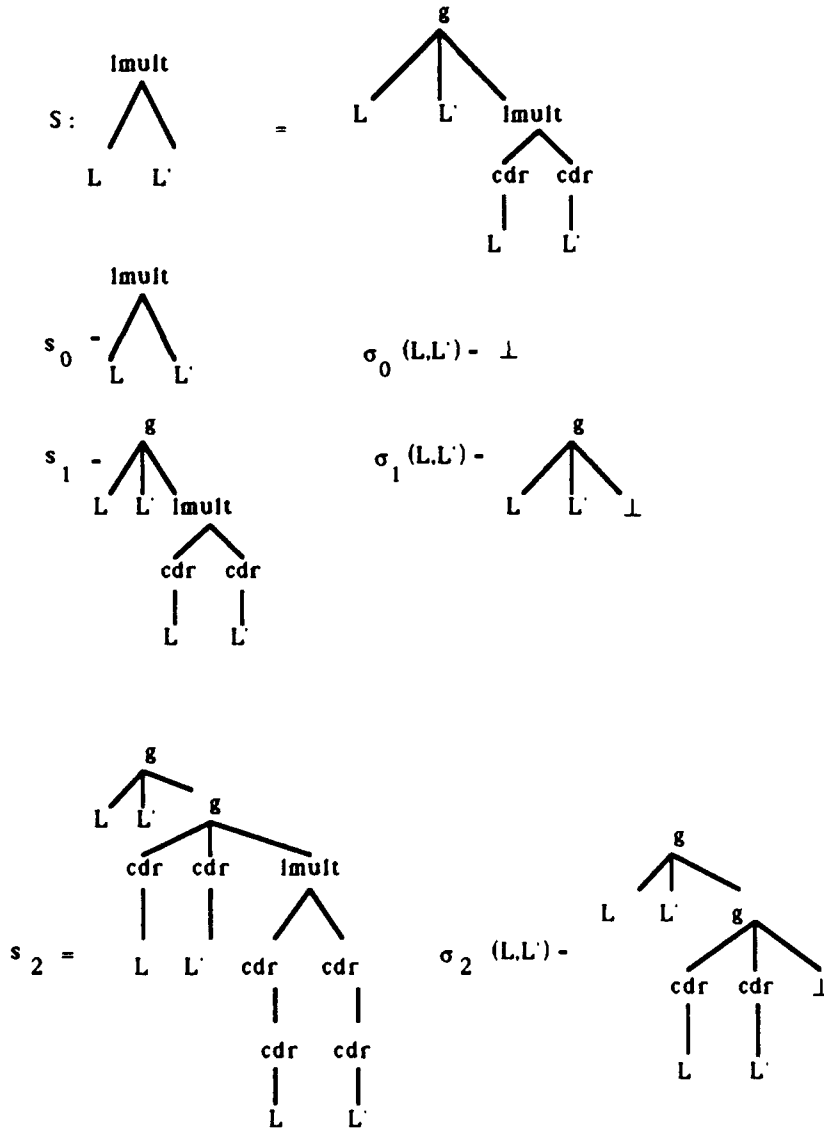


Figure 2.

Finally, the algebraic tree $T(S, lmult(L, L')) = \text{lub}\{\sigma_n / n \in \mathbb{N}\}$ can be depicted as:

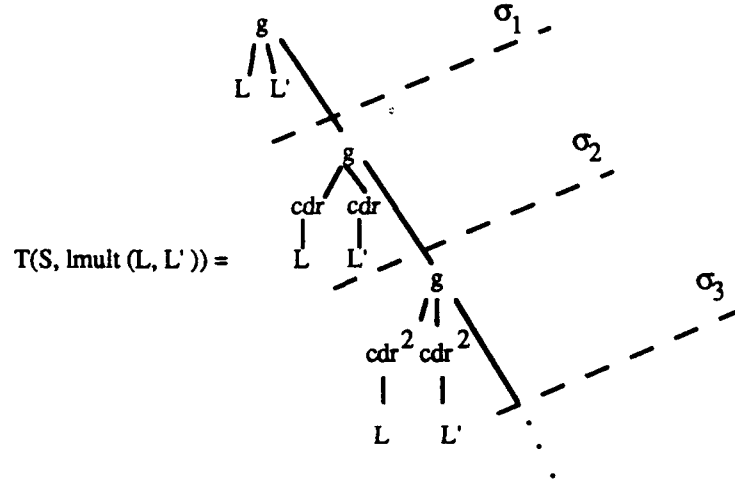


Figure 3.

II.3 Semantics

The basic idea of algebraic semantics is to characterize the meaning of a scheme in the free algebra $CT_{\Sigma}(X)$ via a tree, and to deduce from that tree the meaning of the scheme, and the corresponding programs, in all other possible models, or interpretations.

An *interpretation* A of Σ is a complete Σ -algebra; a *valuated interpretation* is an interpretation together with a valuation $a : X \rightarrow A$. An n -tuple $(\gamma_1, \dots, \gamma_n)$ of operations $\gamma_i : A^{n_i} \rightarrow A$ is said to be a solution of S iff the equations of S are satisfied in A endowed with the $\Sigma \cup \{\phi_1, \dots, \phi_n\}$ -algebra structure defined by $\phi_i A = \gamma_i$ for $i = 1, \dots, n$; equivalently, $(\gamma_1, \dots, \gamma_n)$ is a fixpoint of the system of equations S in A . Note that $H = CT_{\Sigma \perp}(X)$ is a particular interpretation of Σ , called the Herbrand model. We then have:

THEOREM II.3 *The n -tuple $(T_1, \dots, T_n) = (T(S, \phi_1(\vec{x})), \dots, T(S, \phi_n(\vec{x})))$ is the least solution, i.e. the least fixpoint, of the system of equations S in the free algebra $H = CT_{\Sigma \perp}(X)$.*

Proof. We begin by a few remarks which we will use later.

Note first that $T(S, t)$ then is the first component of the least solution of the system $S' = S \cup \{\phi'(\vec{x}) = t\}$ in $CT_{\Sigma \perp}(\vec{x})$, where \vec{x} is the vector of the variables occurring in t .

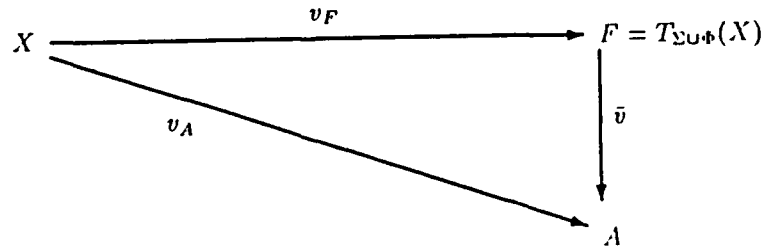
Let A be a complete algebra, and S a system of recursive equations:

$$S : \phi_i(x_1^i, \dots, x_{n_i}^i) = t_i, \quad i = 1, \dots, n \quad (1)$$

Let $A_i = [A^{n_i} \rightarrow A]$ be the set of continuous applications from A^{n_i} into A , for $i = 1, \dots, n$, and $D = A_1 \times \dots \times A_n$.

Fact 1: $A_i, i = 1, \dots, n$, and D are complete Σ -algebras.

Each $\vec{\phi} = (\gamma_1, \dots, \gamma_n) \in D$ determines a $\Sigma \cup \Phi$ -algebra structure on A , where $\Phi = \{\phi_1, \dots, \phi_n\}$ and the γ_i 's are intended to represent the interpretation of the ϕ_i 's in the $\Sigma \cup \Phi$ -algebra, whence a mapping: $t_{iA(\vec{\phi})} : A^{n_i} \rightarrow A$ associating to each valuation $v = (a_1, \dots, a_{n_i}) \in A^{n_i}$ the unique value of $\bar{v}(t_i)$ given by the commutative diagram:



$t_{iA(\vec{\phi})}$ is continuous, hence belongs to A_i . One can thus define a mapping $S_A : D \rightarrow D$ by $S_A(\vec{\phi}) = (t_{1A(\vec{\phi})}, \dots, t_{nA(\vec{\phi})})$.

Fact 2: S_A is continuous. Note that Fact 2 is the basic step in the definition of denotational semantics.

Hence S_A has a least fixpoint μS_A on D , and moreover, μS_A can be computed by $\mu S_A = \text{lub}\{S_A^n(\perp)/n \in \mathbb{N}\}$, where $\perp \in A_i$ is the function everywhere equal to \perp . (Recall that every continuous function f on a complete lattice has a least fixpoint $\mu f = \text{lub}\{f^n(\perp)/n \in \mathbb{N}\}$).

When $A = CT_{\Sigma, \perp}(X)$, one can check that $\text{lub}\{S_A^n(\perp)/n \in \mathbb{N}\} = (T(S, \phi_1(\vec{x})), \dots, T(S, \phi_n(\vec{x})))$, which proves Theorem II.3.

□

Note that: (i) we identify $T \in H = CT_{\Sigma, \perp}(X)$ with T_H , and (ii) for any t , $L(S, t)$ can be obtained by substituting $L(S, t_i)$ to all occurrences of ϕ_i in t , for $i = 1, \dots, n$. By definition, $T(S, t)$ is the function computed by the scheme (S, t) in the free model, or interpretation, H . If A is now an arbitrary interpretation, the function defined by scheme S in A will be defined as $T(S, t)_A$. The adequacy of this definition follows from the:

THEOREM II.4 Let A be a complete Σ -algebra, and (S, t) and (S', t') be two schemes:

- (i) $T(S, t)_A \leq T(S', t')_A$ for all A iff $T(S, t) \leq T(S', t')$
- (ii) for all A , $T(S, t)_A = \text{lub}\{t_k / t_k \leq T(S, t) \text{ and } t_k \in T_{\Sigma, \perp}(X)\}$
- (iii) (T_1, \dots, T_n) is the least solution of S in A .

Theorem II.4 is an immediate consequence of Theorem II.3, together with the freeness of H .

(i) shows that $T(S, t)$ characterizes the behavior of (S, t) with respect to all models, namely that what happens in the model H suffices to describe what will happen in all other models; the name of Herbrand model comes from that property, together with the fact that H is a term model.

(ii) says that the function computed by (S, t) in A can be defined as a lub of finite computations, by successive approximations.

(iii) finally expresses the link between the algebraic semantics above described and the denotational semantics of [Scott].

Theorem II.4 (i) is fundamental in the study of abstract data types [Goguen-Meseguer]: it expresses the fact, that, due to its initiality, the free complete algebra provides an abstract (in the sense that it is representation independent and unique up to isomorphism) way of studying data types. The initial model $CT_{\Sigma, \perp}(X)$ is the abstract data type, wherefrom all other data types can be deduced by homomorphism.

II.4 Nondeterminism

DEFINITION II.5 A non deterministic (as opposed to the deterministic ones considered up to now) recursive scheme on Σ is a pair (S, t) , where S is a system of n equations:

$$S: \quad \phi_i(x_1^i, \dots, x_n^i) = T_i \quad , \quad i = 1, \dots, n \quad (2)$$

where, for $i = 1, \dots, n$, $\phi_i \in \Phi$, $x_j^i \in X$ for each j , $T_i \subseteq T_{\Sigma \cup \Phi}(X)$, and $t \in T_{\Sigma \cup \Phi}(X)$.

(S, t) is said to be iterative if all the trees in the T_i 's and t are left-linear.

Similarly, each scheme S (recursive or iterative) is associated with a tree grammar defined by:

$$S_{\perp}: \quad \phi_i(x_1^i, \dots, x_n^i) \rightarrow T_i + \perp \quad , \quad i = 1, \dots, n$$

As previously, \Rightarrow_S and $\xRightarrow{*}_S$ denote the immediate rewriting according to S_{\perp} and its reflexive and transitive closure.

The solution of the scheme (S, t) in $CT_{\Sigma}(X)$, where X is the set of variables occurring in t , is the forest:

$$F(S, t) = \text{Fin}(S, t) \cup \text{Inf}(S, t)$$

where:

$$\text{Fin}(S, t) = \{t'/t' \in T_{\Sigma}(X) \text{ and } t \xRightarrow{*}_S t'\} = L(S_{\perp}, t)$$

and

$$\begin{aligned} \text{Inf}(S, t) = \{T'/T' \in CT_{\Sigma}(X) \text{ and } T' = \text{lub}\{t_n/n \in \mathbb{N}\} \text{ for some sequence } t_n = t'_n(\vec{\perp}/\vec{\phi}) \\ \text{with } t'_0 = t \text{ and } \forall n \ t'_n \xRightarrow{*}_S t'_{n+1}\} \end{aligned}$$

$\text{Fin}(S, t)$ (resp. $\text{Inf}(S, t)$) correspond to the finite (resp. infinite) trees over Σ generated by the scheme (S, t) , i.e. the finite (resp. infinite) unfoldings of t according to S . Note that no occurrences of \perp appear in the trees of $F(S, t)$.

Note that:

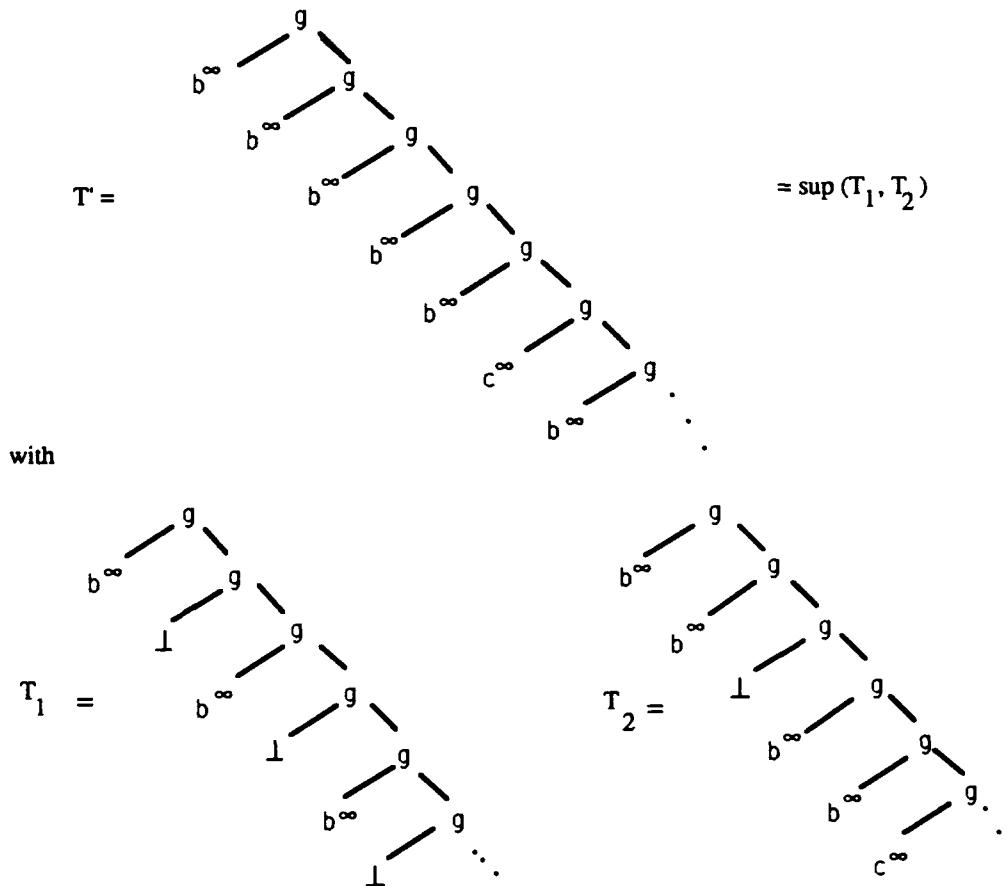
$$\begin{aligned} \text{Inf}(S, t) &\neq \{T'/T' \in CT_E(X) \text{ and } T' = \text{lub}\{t_n/n \in \mathbb{N}\} \text{ for some sequence } t_n \in L(S_\perp, t)\} \\ &= \text{adh}(L(S_\perp, t)) \end{aligned}$$

where $\text{adh}(L)$ is the set of limits of sequences of trees in L . For word grammars we would have $\text{Inf}(S, w) = \text{adh}(L(S, w))$; this shows one respect in which tree languages can be more complex than word languages.

For instance, let S be the iterative scheme:

$$S : \begin{cases} X = g(Y, g(Z, X)) + g(Y, g(Y, g(Z, X))) \\ Y = b(Y) \\ Z = c(Z) \end{cases}$$

Then, letting:



$T' \in \text{adh}(L(S_\perp, X))$, but $T' \notin \text{Inf}(S, X)$ since T' is not a limit of a computation sequence of X according to scheme S , even though $T' = \text{sup}(T_1, T_2)$ with T_1 and T_2 partial limits of (non terminated) computation sequences of X according to S , i.e. $\exists T'_i \in \text{Inf}(S, X), T_i \leq T'_i, i = 1, 2$.

EXAMPLE II.6 Consider the recursive scheme:

$$S : \phi(x) = x + a(\phi(b(x)))$$

$(S, \phi(x))$ generates the algebraic forest, which happens here to be identifiable with a word language: $F(S, \phi(x)) = \{a^n b^n(x)/n \geq 0\} \cup \{a^\omega\}$, where a^ω denotes the infinite tree $a^\omega = \text{lub}\{a^n(\perp)/n \geq 0\}$. Here $\text{Inf}(S, \phi(x)) = a^\omega$.

We can thus see that the case of the deterministic program schemes generating a single infinite tree corresponds to schemes whose finitary part is empty and whose infinitary part is reduced to a single element.

Remark. Theorem II.3 is no longer true in the case of non deterministic schemes. For instance, in Example II.6, $\{a^n b^n(x)/n \geq 0\} \cup \{a^\omega\}$ is the greatest fixpoint of the system of equations S in $CT_\Sigma(X)$; note that $CT_\Sigma(X) = \{a, b\}^\omega(x)$. Examples can be found where $F(S, \phi(\vec{x}))$ is neither the greatest nor the least fixpoint of S in the corresponding free algebras [Guessarian 89]. This point shows the advantage of the approach of Section II.2, where all schemes were "determinized" by considering "+" and "or" as just formal uninterpreted symbols of rank 2 in Σ , and where we solved the schemes in the free algebra $CT_{\Sigma_\perp}(X)$, by comparison to the "more natural" approach in the non-deterministic case, where the non-deterministic "or" is explicitly interpreted as set union. In the latter case, the natural free domains are $\mathcal{P}(CT_\Sigma(X))$, equipped with set inclusion as ordering, or the closed subsets thereof [Arnold-Nivat], and then, $F(S, \phi(\vec{x}))$ is no longer necessarily a component of the least fixpoint of S in the free Σ -algebra, as shown by the above examples. We will use this remark in section IV to help us when giving the semantics of concurrent processes.

III APPLICATION TO LOGICS OF PROGRAMS

III.1 Inequational logics

We now show how to translate Theorem II.4 (i) in terms of many sorted inequational logics. Notice first that $T_A \leq T'_A$ for all interpretations A means that $T \leq T'$ is *semantically* valid, i.e. that $\models T \leq T'$ (we more generally take arbitrary infinite trees T, T' instead of $T(S, t)$ and $T(S', t')$). Note then that $T \leq T'$ amounts to *syntactic* provability in a system of deduction rules expressing that \leq is an ordering. Hence Theorem II.4 (i) can be viewed as a Birkhoff-like completeness theorem for proving all valid inequations: formally

THEOREM III.1 For T, T' in $CT_{\Sigma_\perp}(X)$,

$$\models T \leq T' \quad \text{iff} \quad Ax \vdash_\infty T \leq T'$$

where:

- Ax is the set of axioms:

$$\perp \leq t \text{ for any } t \text{ in } CT_{\Sigma_\perp}(X), \quad (3)$$

- \vdash_∞ is the following set of deduction rules:

$$t \leq t \quad \text{for any } t \text{ in } T_{\Sigma_\perp}(X) \quad (\text{reflexivity}) \quad (4)$$

$$t \leq t', t' \leq t'' \vdash t \leq t'' \quad \text{for any } t, t', t'' \text{ in } T_{\Sigma_\perp}(X) \quad (\text{transitivity}) \quad (5)$$

$$\begin{aligned} &\text{for } t_i, t'_i \text{ in } T_{\Sigma_\perp}(X), i = 1, \dots, n, \quad \sigma \in \Sigma \\ &\quad \forall i = 1, \dots, n \quad t_i \leq t'_i \quad \vdash \quad \sigma(t_1, \dots, t_n) \leq \sigma(t'_1, \dots, t'_n) \end{aligned} \quad (\text{monotonicity}) \quad (6)$$

$$\forall j \in \mathbb{N} \quad \exists i \in \mathbb{N} \quad t_i \leq t'_j \quad \vdash_\infty \quad \text{lub}_{i \in \mathbb{N}}(t_i) \leq \text{lub}_{j \in \mathbb{N}}(t'_j) \quad \text{for } t_i, t'_j \text{ in } T_{\Sigma_\perp}(X) \quad (\text{continuity}) \quad (7)$$

Theorem III.1 states that the axiom system Ax together with the deduction rules \vdash_∞ is complete for proving all valid inequalities. Note then, that since $\models T = T'$ iff $(\models T \leq T' \text{ and } \models T' \leq T)$, the previous theorem also provides us with a complete proof system for proving equalities. Deduction systems using the rules of equational logic for proving equalities only would also be possible. We will see such an example later.

III.2 Classes of interpretations

Usually, we do not consider all models (or interpretations) but only subclasses of models subject to some constraints, e.g. that some operation σ is a test, that some other operations are associative, or commutative, etc... Given a class \mathcal{C} of models, we would thus like to find an "abstract" or Herbrand model which characterizes the behaviour of the whole class \mathcal{C} . In other words, we are trying to generalize theorems II.4 (i) and III.1 of the previous sections, by finding a model $H_{\mathcal{C}}$ such that

$$\mathcal{C} \models T \leq T' \quad \text{iff} \quad H_{\mathcal{C}} \models T \leq T' \quad (8)$$

We would also like to find a complete system of axioms $Ax_{\mathcal{C}}$ and deduction rules \vdash' such that:

$$\mathcal{C} \models T \leq T' \quad \text{iff} \quad Ax_{\mathcal{C}} \vdash' T \leq T' \quad (9)$$

This leads to considering factor algebras of the algebras $T_{\Sigma}(X)$, CT_{Σ} , and usually is the source of many problems, because the factor algebra is no longer complete, or, even if it is complete, continuity is not preserved, etc... See [Courcelle, Guessarian, Courcelle-Guessarian] for some of the problems, and references to related literature.

However, there is one case when we obtain a nice characterization of the free and Herbrand algebra relative to the class \mathcal{C} , generalizing the construction of CT_{Σ} , this is the case of relational (or equational) classes. In these cases, both equivalences (8) and (9) can be satisfied quite easily. Moreover, equational classes are among the most widely used in abstract data type theory.

DEFINITION III.2 A class \mathcal{C} of models (i.e. interpretations or Σ -complete algebras) is said to be relational iff it is of the form:

$$\mathcal{C}_R = \{A/A \models t \leq t', \text{ for all } (t, t') \text{ in } R\},$$

where $R \subseteq T_{\Sigma, \perp}(X)^2$. \mathcal{C}_R is said to be equational if R is an equivalence relation. In other words, an equational class is of the form $\mathcal{C}_R^e = \{A/A \models t = t', \text{ for all } t, t' \text{ in } R\}$, where R is as above.

Relational classes form semi-varieties and have been considered by different authors under various names (see [Guessarian] for references concerning the subject). Let \prec_R be the least substitution-closed (or fully invariant) preordering containing R on $T_{\Sigma, \perp}(X)$ (i.e. $t \prec_R t'$ implies that for any homomorphism $h : T_{\Sigma, \perp}(X) \rightarrow T_{\Sigma, \perp}(X)$, we also have $h(t) \prec_R h(t')$). Let \sim_R be the associated equivalence relation. Define $H_R = T_{\Sigma, \perp, R}(X)^{\infty}$, where $T_{\Sigma, \perp, R}(X)$ is the ordered Σ -algebra obtained by factoring $T_{\Sigma, \perp}(X)$ through \sim_R and ordering it with \prec_R / \sim_R , and $T_{\Sigma, \perp, R}(X)^{\infty}$ is the ideal completion of $T_{\Sigma, \perp, R}(X)$.

THEOREM III.3 Let $R \subseteq T_{\Sigma, \perp}(X)^2$, and $H_R = T_{\Sigma, \perp, R}(X)^{\infty}$. Then H_R is an initial and Herbrand model for the class \mathcal{C}_R , i.e. for T, T' in $CT_{\Sigma, \perp}(X)$:

$$\mathcal{C}_R \models T \leq T' \quad \text{iff} \quad H_R \models T \leq T' \quad (10)$$

For the proof, see [Guessarian].

As in the previous subsection, we can immediately translate this theorem into a completeness theorem for the inequational logics of classes of the form \mathcal{C}_R .

THEOREM III.4 For T, T' in $CT_{\Sigma, \perp}(X)$,

$$\mathcal{C}_R \models T \leq T' \quad \text{iff} \quad Ax_R \vdash'_{\infty} T \leq T' \quad (11)$$

where:

- Ax_R is the set of axioms:
 - $\perp \leq t$ for any t in $CT_{\Sigma, \perp}(X)$,
 - $t \leq t'$ for any (t, t') in R .
- \vdash'_{∞} is the set (4-7) of deduction rules previously stated, together with the rule:

$$t \leq t' \vdash h(t) \leq h(t'), \text{ for any } t, t' \text{ in } T_{\Sigma, \perp}(X), \text{ and endomorphism } h : T_{\Sigma, \perp}(X) \rightarrow T_{\Sigma, \perp}(X). \quad (12)$$

The deduction rule (12), expressing full-invariance or substitution closure, was not needed for the Theorem III.1 because it was ipso facto satisfied by the ordering \leq on $CT_{\Sigma}(X)$, and introduced no new terms in this ordering. It is needed now, because we have new axioms in Ax_R , those coming from the relation R , which is not necessarily substitution-closed, while on the other hand the validity relation $\mathcal{C}_R \models t \leq t'$ is obviously substitution-closed.

Theorem III.4 has numerous applications in abstract data types, proofs of program properties, description of the semantics of parallel languages like CCS [Milner] or ACP [Bergstra-Klop], construction of fully abstract models for such languages, etc...

III.3 An example of equational logics

The following example shows how we can obtain a complete system of axioms and inference rules for proving all valid equations, even though the considered class of interpretations is not (in)equational. See [Bloom-Tindell, Guessarian-Meseguer] for more details.

Let Σ be the signature: $\Sigma = \langle \perp, \#, \text{ff}, g \rangle$ where $\perp, \#, \text{ff}$ have rank 0, and g is a rank 3 operation; let \mathcal{C} be the class of algebras such that:

$$g_A(p, x, y) = \begin{cases} x & \text{if } p = \# \\ y & \text{if } p = \text{ff} \\ \perp & \text{if } p \neq \#, \text{ff} \end{cases}$$

The class \mathcal{C} not being closed under products is neither equational nor inequational. However, we have the:

THEOREM III.5 *For any infinite terms t, t' in H , $\mathcal{C} \models t = t'$ iff $Ax \vdash_{=} t = t'$ where $\vdash_{=}$ is deduction using the rules of equational logic (i.e. computing the congruence relation generated by Ax , or equivalently, the rules expressing reflexivity, symmetry and transitivity). Ax is the following axiom system.*

$$\begin{aligned} [\#, x, y] &= x; & [p, p, x] &= [p, \#, x] \\ [\text{ff}, x, y] &= y; & [p, x, p] &= [p, x, \text{ff}] \\ [\perp, x, y] &= \perp; & [p, \perp, \perp] &= \perp \\ [p, [p, x, y], z] &= [p, x, z]; & [p, x, [p, y, z]] &= [p, x, z] \\ [p, [q, x, y], [q, u, v]] &= [q, [p, x, u], [p, y, v]] \\ [[p, x, y], u, v] &= [p, [x, u, v], [y, u, v]] \end{aligned}$$

(where $[p, x, y]$ abbreviates $g_A(p, x, y)$).

See [Bloom-Tindell, Guessarian-Meseguer] for more details. The algebras in \mathcal{C} are one-sorted, but this result can be extended to many-sorted algebras.

IV APPLICATION TO SEMANTICS OF CONCURRENCY

IV.1 Fundamentals of fixpoint semantics

We first recall here the classical Knaster-Tarski fixpoint theorem, in its constructive variant, most widely used in computer science.

THEOREM IV.1 (Fixpoint theorem) *Let E be a complete Σ -algebra, and $f : E \rightarrow E$ a continuous function; then f has a least fixpoint μf such that $f(\mu f) = \mu f$, and for all $e \in E$, $f(e) = e \Rightarrow e \geq \mu f$. Moreover, μf is computed by $\mu f = \sup\{f^n(\perp) / n \in \mathbb{N}\}$.*

This theorem provides the basis of most of the work done in semantics, algebraic semantics [Nivat, ADJ], denotational semantics [Scott, Rounds], logic programming [Van Emden-Kowalski].

The basic idea then is the following: given a program scheme S consisting of a set of recursive equations on a base signature Σ , we first solve S syntactically using Theorem IV.1 in the Herbrand model, namely the free continuous Σ -algebra CT_{Σ} consisting of finite and infinite well formed Σ -trees. We thus obtain a generally infinite tree $T(S)$ which is the least fixpoint of S . We then interpret $T(S)$ in an arbitrary model A , by taking its image under a strict continuous morphism ϕ , which gives us the semantics of S in the model A ; this image is the least fixpoint of the program $\phi(S)$ in the model A . The situation is illustrated by the following example.

EXAMPLE IV.2 Let $S : F(n) = g(n, F(p(n)))$, where $\Sigma = \{\perp, p, g\}$. Then $T(S)(n) = g(n, g(p(n), g(p^2(n), \dots))) = \sup\{g(n, \perp), g(n, g(p(n), \perp)), \dots\}$. Consider now as model $A = \mathbb{N} \cup \{\perp_A\}$, with the discrete ordering having least element \perp_A . Define $p_A(n) = n - 1$ and $g_A(n, m) =$ if $n = 0$ then 0 else $n + m$, where $+$ and $-$ are the usual operations on \mathbb{N} , and all functions are suitably extended to deal with \perp_A . Then $\phi(S)$ is the recursive definition $f(n) =$ if $n = 0$ then 0 else $n + f(n - 1)$. Let $\phi : CT_{\Sigma} \rightarrow A$ be the morphism defined by $\phi(\sigma) = \sigma_A$ for σ in Σ ; we have $\phi(T(S))(n) = n(n + 1)/2$, and $n \mapsto n(n + 1)/2$ is indeed the least fixpoint of $\phi(S)$ in A , and provides us with the intended meaning of $\phi(S)$ in A .

More generally we have the following, stated for simplicity in the case of a single equation, but which holds for an arbitrary system of recursive equations.

PROPOSITION IV.3 Let Σ be a signature, $S : F = t(F)$ a recursive equation where $t(F)$ is a well-formed term on the signature $\Sigma \cup \{F\}$. Let A be a complete Σ -algebra and ϕ the morphism $\phi : CT_\Sigma \rightarrow A$ defined by: $\phi(\sigma) = \sigma_A$ for σ in Σ . Let $\phi(S)$ be the recursive equation $F = t_A(F)$, where t_A is deduced from t by substituting the σ_A 's for the σ 's. Let $\mu S'$ denote the least fixpoint of S' for S' in $\{S, \phi(S)\}$, then $\phi(\mu S) = \mu(\phi(S))$.

The proof is an immediate consequence of the freeness of CT_Σ . We identify the least solution of S with the corresponding least fixpoint. μS gives a syntactic description of the computations of S in the free model CT_Σ ; to have an effective description of the computations in an actual model A , it is enough to use Proposition IV.3, and take the image of μS under the morphism $\phi : CT_\Sigma \rightarrow A$.

IV.2 The parallel case

Unfortunately, for parallel or nondeterministic programs, the image of μS under ϕ does not necessarily yield the computations we are looking for as shown by the:

EXAMPLE IV.4 Let $S : F = a \cdot F + \tau \cdot F$ be a recursive definition on the signature $\Sigma = \{\perp, a, \tau, +\}$ and let $B = a^* \cup \{a^\omega\}$. Consider the algebra $A = \mathcal{P}(B)$, ordered by inclusion, with $\perp_A = \emptyset$, $\tau_A = id$, $+_A = \cup$, and $a_A(C) = \{a \cdot c / c \in C\}$. Then μS is the rational tree T defined by $T = a \cdot T + \tau \cdot T$ in CT_Σ ; in the algebra A , however, we obtain $\phi(T) = \emptyset$, which is indeed the least fixpoint of $\phi(S) : F = a \cdot F \cup F$. This may be justified when dealing with deterministic programs, because $\phi(S)$ represents a loop; but it is no longer acceptable if we wish to model parallel and nondeterministic programs; in this latter case, τ would represent an invisible move, but we would nevertheless be interested in keeping some track of the infinite sequence of actions a . So, following [Arnold-Nivat] we would in the present case define the intended meaning of $x = ax$ in A as being its greatest fixpoint $\{a^\omega\}$ instead of its least fixpoint \emptyset . Similarly, the semantics of $x = ax \cup x$ in A will be its greatest fixpoint $a^* \cup \{a^\omega\}$ instead of its least fixpoint \emptyset . The same semantics is obtained in [Rounds] with slightly different tools.

Not even greatest fixpoints solve all problems, however:

EXAMPLE IV.5 Consider the same signature as in Example IV.4, and let $S : F(x) = a \cdot x + F(a \cdot x)$. Let A be defined as in Example IV.4, then the greatest fixpoint of $\phi(S) : F(x) = a \cdot x \cup F(a \cdot x)$ is $a^+ \cup \{a^\omega\}$, and there has been some argumentation in the literature as to whether this might be attributed as semantics to $\phi(S)$ (see [Broy, Arnold-Nivat] for conflicting opinions). We consider that a^ω , which is *not* a limit of partial computations of $\phi(S)$, should therefore not be incorporated to its semantics, for the solace of preserving greatest fixpoints.

Combining Example IV.4 and Example IV.5 shows that arbitrary fixpoints might be needed.

The problem, as illustrated by the previous examples, is to find a suitable way to translate the least fixpoint in the free model into the "right" fixpoint in the model A . The "right" fixpoint is supposed to take care of finite as well as infinite computations in A . So this problem amounts to studying alternate ways of transforming fixpoints by morphisms. Solutions have been proposed to this problem, but they basically amount to considering a case when the fixpoint is unique, as in the case of Greibach schemes [Arnold-Nivat], or guarded schemes [Bergstra-Klop]. These ideas will not apply because the scheme $\phi(S)$ will not be Greibach or guarded (Example IV.5), or the operator corresponding to $\phi(S)$ will not be contracting in A (Example IV.4). Example IV.4 also shows that we will not always be able to transform a least fixpoint into a greatest one via a closure operator: in that example, the closure of the empty set would have to be $a^* \cup \{a^\omega\}$, which is too demanding.

In the rest of this section, we will address an instance of the problem of transforming least (or unique) fixpoints via morphisms, which will be tailored for describing the semantics of communicating processes. Our formalism will be inspired from CCS (or ACP) - like languages [Milner, Bergstra-Klop]. We will adopt a modular approach, and consider first the nondeterministic case, and then the concurrent case, that we will try to solve using the results obtained in the nondeterministic case.

IV.3 Fixpoints and morphisms for nondeterminism

Let A be an alphabet of unary action symbols, let NIL be a constant symbol representing deadlock (or termination), let τ be a distinguished symbol representing an invisible action and $+$ be a binary symbol (representing nondeterministic choice). Let Σ be the signature $\Sigma = \Sigma_1 = \langle NIL, A, \tau, + \rangle$, and let T_Σ (resp. CT_Σ) denote the set of finite (resp. finite and infinite) Σ -trees.

Let $S : F(x) = t(F)(x)$ be a recursive equation, with $t(F)(x) \in T_{\Sigma \cup \{F, x\}}$; t is thus a finite term on the signature Σ . For simplicity of notations we will assume that we have a single recursive equation, the case of several mutually recursive equations would be similar. S is said to be *weakly guarded* (or weakly Greibach) iff it is not of the form $F(x) = F(t'(x))$.

PROPOSITION IV.6 *Let $S : F(x) = t(F)(x)$ be weakly guarded, then S has a unique fixpoint $T(S)$ in CT_Σ . $T(S)$ is thus also the least fixpoint of the operator $\lambda F.t(F)$, with the notations of [Rounds].*

The scheme S is recursive whereas for modelling CCS (or ACP) like processes we need only rational schemes of the form $F = t(F)$. We will keep this more general framework as long as it will not cost us any extra work. Our goal will be to use $T(S)$ to give the semantics of S in the classical model $\mathcal{P}(A^\omega)$ consisting of languages of finite and infinite words on the alphabet A , in a way which would render a good account of its operational semantics. We would like to define ϕ so as to obtain a mapping:

$$\phi : CT_\Sigma \longrightarrow \mathcal{P}(A^\omega) \quad (13)$$

where $\phi(T(S))$ would represent all the computations (i.e. possible finite or infinite sequences of actions of $T(S)$). Moreover, the semantics of NIL , $+$, τ in $\mathcal{P}(A^\omega)$ should satisfy:

$$\begin{aligned} \phi(NIL) &= \varepsilon, \quad \phi(T + T') = \phi(T) \cup \phi(T'), \\ \phi(aT) &= a \cdot \phi(T), \quad \phi(\tau T) = \phi(T). \end{aligned} \quad (14)$$

One may try and define ϕ

- inductively for finite trees by the equations (14)
- by "continuity" for infinite trees by:

$$\varphi_1(\sup\{t_n/n \in \mathbb{N}\}) = \sup\{\phi(t_n)/n \in \mathbb{N}\}$$

or by:

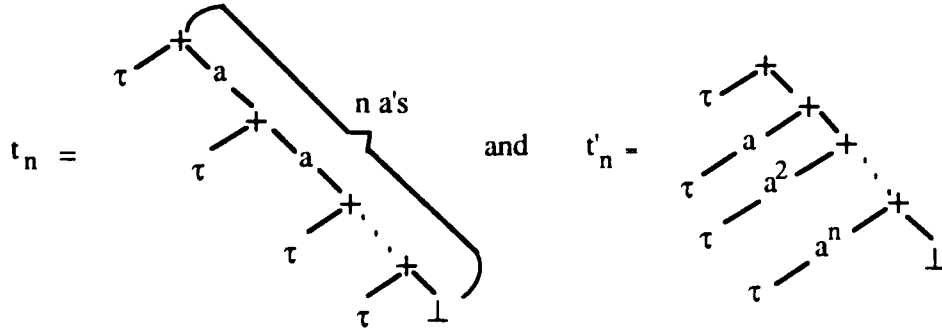
$$\varphi_2(\sup\{t_n/n \in \mathbb{N}\}) = adh(\sup\{\phi(t_n)/n \in \mathbb{N}\})$$

Recall that:

DEFINITION IV.7 *For L a language in $\mathcal{P}(A^\omega)$, $adh(L)$ is the set of infinite words $w \in A^\omega$ such that all left factors of w belong to the set of left factors of L .*

But then neither φ_1 nor φ_2 give the desired semantics for infinite trees as shown by the following example:

EXAMPLE IV.8 Consider the trees:



the \perp 's are present for technical soundness reasons which will be justified in the next section. We obtain by (14):

$$\phi(t_n) = \phi(t'_n) = \{\varepsilon, a, \dots, a^n\}$$

Letting: $T = \sup\{t_n/n \in \mathbb{N}\}$ and $T' = \sup\{t'_n/n \in \mathbb{N}\}$, we will thus obtain, by continuity:

- $\varphi_1(T) = \varphi_1(T') = a^*$, which does not give the right semantics for T , since a^ω should belong to $\varphi_1(T)$

- $\varphi_2(T) = \varphi_2(T') = a^* \cup \{a^\omega\}$, which does not give the right semantics for T' , since a^ω should not belong to $\varphi_2(T')$.

If parallel composition operators are present, the problems might even get worse, because ϕ , as defined by the equations (14) might even be non monotone, rendering impossible an extension by continuity.

We will consider here the case when $\Sigma = \Sigma_1 = \langle NIL, A, \tau, + \rangle$. For T in CT_{Σ_1} , define $br(T)$ the set of maximal paths in T , i.e. the root to leaf (or infinite) sequences of pairs of node labels and node numbers along a path in T . We will thus have, for b^1, \dots, b^n , in Σ_1 , and $1, \dots, n$ in \mathbb{N}^* : $(b^1, 1) \dots (b^n, n) \dots \in br(T)$ iff $1 \dots n \dots$ is a maximal path in T and $b^n = T(n)$ is the label of node n in T . We can formally define $br(T)$ by induction on k , by letting $T = \sup\{t_k/k \in \mathbb{N}\}$, the t_k 's being finite trees ordered by the relation "to be an initial subtree of".

Let then $\pi_A : \Sigma_1 \times \mathbb{N}^* \rightarrow A$ be the projection morphism defined by:

$$\pi_A(b^n, n) = \begin{cases} b^n, & \text{if } b^n \in A; \\ \varepsilon, & \text{if } b^n \in \{\tau, +\}. \end{cases}$$

Define finally: $\phi(T) = \pi_A(br(T))$. $\phi(T)$ will provide us with the required mapping for the case when $\Sigma = \Sigma_1$. ϕ clearly satisfies the morphism requirements (14), and consequently defines an adequate semantics for finite trees in T_{Σ_1} . Moreover, ϕ also defines a suitable semantics for infinite trees (the reader is invited to check ϕ against Example IV .8).

THEOREM IV.9 Let

$$S : T = t(T) \quad (15)$$

be a left linear "à la CCS" recursion and $T(S)$ be the least fixpoint of S in CT_{Σ_1} . Then:

(i) $\phi(T(S))$ is a fixpoint of the equation

$$X = \phi[t](X) \quad (16)$$

on $\mathcal{P}(A^\infty)$, where $\phi[t] : \mathcal{P}(A^\infty) \rightarrow \mathcal{P}(A^\infty)$ is defined by induction on the depth of t by, for $X \subseteq \mathcal{P}(A^\infty)$:

$$\phi[t](X) = \begin{cases} X & \text{if } t(X) = X, \\ \varepsilon & \text{if } t(X) = NIL, \\ \phi[t_1](X) \cup \phi[t_2](X) & \text{if } t = t_1 + t_2, \\ a \cdot \phi[t_1](X) & \text{if } t = at_1, \\ \phi[t_1](X) & \text{if } t = \tau t_1. \end{cases}$$

(ii) If moreover $\phi[t](X)$ is weakly guarded (or Greibach), then $\phi(T(S))$ is the greatest fixpoint of (16).

A term $\theta = \bigcup_{i=1}^n w_i X \cup U$ is said to be weakly guarded if at least one of the w_i 's is different from ε , i.e. if θ is not of the form $X \cup U$. t is applied to (possibly infinite) terms in CT_{Σ_1} , whereas $\phi[t]$ is applied to (possibly infinite) sets of words in $\mathcal{P}(A^\infty)$. With the notations of [Guessarian], t would be denoted by $t_{CT_{\Sigma_1}}$, whereas $\phi[t]$ would be denoted by $t_{\mathcal{P}(A^\infty)}$.

Sketch of proof: (i) (15) will be translated into an equation involving $br(T)$, to which we will apply π_A , and this will yield the equation (16) we are looking for, namely $\phi(T) = \phi[t](\phi(T))$. ϕ is not continuous, but continuity is not needed because we only deal with a finite term t , and the corresponding $\phi(t)$ is also finite.

The proof of (ii) is more complicated, and needs an induction on the structure of t and intermediate results.

Our Theorem IV .9 also holds if S is a system of mutually recursive equations of the form:

$$S : \begin{cases} T_1 = t_1(T_1, \dots, T_k) \\ \vdots \\ T_k = t_k(T_1, \dots, T_k) \end{cases} \quad (17)$$

Thus, for systems S of the form (15) or (17), on the signature $\Sigma_1 = \langle NIL, A, \tau, + \rangle$, Theorem IV .9 enables us to deduce the computations of S in an arbitrary model A from its syntactic computation in the free model CT_{Σ_1} , via the morphism ϕ defined by $\phi(T) = \pi_A(br(T))$. This yields, in that case, a clean semantics for parallel programs, based on initial algebras and morphisms.

IV.4 Extension to true concurrency

The case when $\Sigma = \Sigma_2 = \langle NIL, A, \tau, +, \parallel \rangle$, where \parallel is a binary symbol representing a parallel composition, e.g. the Δ -synchronized shuffle of [Rounds], where actions in Δ are synchronized, and actions out of Δ are interleaved, is more complex. We will adopt a modular approach to the problem, and try to reduce it to the nondeterministic case.

For technical reasons, we will introduce in our signatures a new symbol \perp representing undefined, or "pending" computations, as in the theory of algebraic semantics [Guessarian], or, more recently, [Aceto-Hennessy].

We will in fact consider the more general case of a signature $\Sigma_2 = \langle NIL, A, \tau, +, \perp \rangle \cup \{o_1, \dots, o_n\}$, where the o_i 's are symbols which can be interpreted as \parallel_Δ [Rounds], \parallel [Milner], \parallel [Bergstra-Klop], the restriction or hiding operators, substitutions or renamings, the sequential composition ";", of [Rounds], etc. . . .

The only constraint will be that each o_i be interpreted in $\mathcal{P}(A^\infty)$ as an operator \bar{o}_i which can be defined inductively by Definition IV .10. For simplicity we state this definition in the case of a single binary operator o , the general case of an operator of arbitrary rank being similar but more tedious.

DEFINITION IV.10 Let \bar{o} be defined on $\mathcal{P}(A^\infty)$ as follows: (i) for $w = a_1 \dots a_n = a_1 u$ and $w' = a'_1 \dots a'_p = a'_1 u'$ two finite words in A^* :

$$\begin{aligned} \bar{o}(\varepsilon, \varepsilon) &= \varepsilon \\ \bar{o}(w, w') &= \bigcup_k \chi_k(a_1, a'_1) \cdot \bar{o}(u, u') + \bigcup_k \theta_k(a_1) \cdot \bar{o}(u, w') + \bigcup_k \theta'_k(a'_1) \cdot \bar{o}(w, u') \end{aligned}$$

This definition implies that: $\bar{o}(w, \varepsilon) = \theta(w)$ where θ is a morphism for concatenation. We will in the sequel write the above definition in the shorthand form:

$$\bar{o}(w, w') = \bigcup_k \chi_k(a_1, a'_1) \cdot \bar{o}(\nu_k(w), \nu'_k(w'))$$

(ii) For w, w' two infinite words in A^∞ ,

$$\bar{o}(w, w') = adh\{\cup\{\bar{o}(u, u')/u \text{ left factor of } w, u' \text{ left factor of } w'\}\}.$$

(iii) For L, L' two languages in $\mathcal{P}(A^\infty)$,

$$\bar{o}(L, L') = \cup\{\bar{o}(w, w')/w \in L, \text{ and } w' \in L'\}.$$

In the sequel and in order to simplify notations, we will assume that there is a single binary operator o satisfying Definition IV .10. The case of several o_i 's of various ranks is similar.

EXAMPLE IV.11 An example of operator o is the Δ -synchronized parallel composition \parallel_Δ which is a combination of Milner's parallel composition and Rounds' [Rounds] Δ -synchronized shuffle: it interleaves actions outside of Δ , and can either interleave or synchronize events in Δ . Formally, it satisfies the following expansion laws:

- if $p = \sum_{i \in I} a_i p_i$ and $q = \sum_{j \in J} b_j q_j$ then:

$$p \parallel_\Delta q = \sum_{i \in I} a_i \cdot (p_i \parallel_\Delta q) + \sum_{j \in J} b_j \cdot (p \parallel_\Delta q_j) + \sum_{a_i = \bar{b}_j \in \Delta} \tau \cdot (p_i \parallel_\Delta q_j) \quad (18)$$

- if $p = \sum_{i \in I} a_i p_i + \perp$ and $q = \sum_{j \in J} b_j q_j$ or $q = \sum_{j \in J} b_j q_j + \perp$, or if $p = \sum_{i \in I} a_i p_i$ and $q = \sum_{j \in J} b_j q_j + \perp$, then:

$$p \parallel_\Delta q = \sum_{i \in I} a_i \cdot (p_i \parallel_\Delta q) + \sum_{j \in J} b_j \cdot (p \parallel_\Delta q_j) + \sum_{a_i = \bar{b}_j \in \Delta} \tau \cdot (p_i \parallel_\Delta q_j) + \perp \quad (18')$$

\bar{o} is then the operator $shuffle_\Delta$ which is defined along the lines of the Definition IV .10 by: (i) For $w = a_1 \dots a_n, w' = a'_1 \dots a'_p$, two finite words in A^* ,

$$\begin{aligned} shuffle_\Delta(\varepsilon, w) &= shuffle_\Delta(w, \varepsilon) = w \\ shuffle_\Delta(w, w') &= a_1 \cdot shuffle_\Delta(a_2 \dots a_n, a'_1 \dots a'_p) \cup a'_1 \cdot shuffle_\Delta(a_1 \dots a_n, a'_2 \dots a'_p) \cup S \end{aligned}$$

where

$$S = \begin{cases} shuffle_\Delta(a_2 \dots a_n, a'_2 \dots a'_p) & \text{if } a_1 = \bar{a}'_1 \in \Delta \\ \emptyset & \text{otherwise.} \end{cases}$$

(ii) For w, w' two infinite words in A^∞ ,

$$\text{shuffle}_\Delta(w, w') = \text{adh}\{\cup\{\text{shuffle}_\Delta(u, u')/u \text{ left factor of } w, u' \text{ left factor of } w'\}\}.$$

(iii) For L, L' two languages in $\mathcal{P}(A^\infty)$,

$$\text{shuffle}_\Delta(L, L') = \cup\{\text{shuffle}_\Delta(w, w')/w \in L, \text{ and } w' \in L'\}.$$

Remark. Milner's parallel composition \parallel corresponds to \parallel_A .

We would like $\psi = \phi \circ \phi_1 : CT_{\Sigma_2} \rightarrow \mathcal{P}(A^\infty)$ to satisfy the equations (14) suitably extended in order to deal with the undefined element \perp and the operator o , i.e.:

$$\begin{aligned} \psi(NIL) &= \varepsilon, \psi(T + T') = \psi(T) \cup \psi(T'), \\ \psi(aT) &= a \cdot \psi(T), \psi(\tau T) = \psi(T), \\ \psi(T o T') &= \bar{o}(\psi(T), \psi(T')), \psi(\perp) = \emptyset. \end{aligned} \quad (14')$$

For the same reason as in Example IV .8, a direct definition of ψ is not possible. The idea then is to decompose $\psi : CT_{\Sigma_2} \rightarrow \mathcal{P}(A^\infty)$ into $\psi = \phi \circ \theta$, where $\theta : CT_{\Sigma_2} \rightarrow CT_{\Sigma_1}$, and $\phi : CT_{\Sigma_1} \rightarrow \mathcal{P}(A^\infty)$; however, θ is neither monotone nor well defined, and $\phi_1 : CT_{\Sigma_2} \rightarrow CT_{\Sigma_1}/\sim$ has to be introduced, where CT_{Σ_1}/\sim is a suitable factor of CT_{Σ_1} . Namely we will try and construct a commutative diagram as shown by Figure 4, where \sim is defined by:

DEFINITION IV.12 (i) Let \sim be the congruence defined on T_{Σ_2} (the finite terms in CT_{Σ_2}) by the axioms:

$$\begin{aligned} \perp \cdot p &\sim \perp \\ (p + q) + r &\sim p + (q + r) \\ p + q &\sim q + p \end{aligned}$$

(ii) \sim is extended to infinite terms $T, T' \in CT_{\Sigma_2}$ by: $T \sim T'$ iff there exist increasing chains t_n and t'_n such that $T = \sup\{t_n/n \in \mathbb{N}\}$, $T' = \sup\{t'_n/n \in \mathbb{N}\}$ and $t_n \sim t'_n$ for all n .

The restriction of the the congruence \sim on T_{Σ_1} (resp. CT_{Σ_1}), is also denoted by \sim .

The above definition is not the standard way of extending \sim to infinite trees, which was used in Theorem III .3 and which consists in:

- 1) Defining the preorder $\leq = (\prec \cup \sim)^*$ (the transitive closure of $\prec \cup \sim$) on T_{Σ_2} .
- 2) Extending \leq to CT_{Σ_2} by letting $T \leq T'$ iff $\forall t \prec T, \exists t' \prec T', T \leq T'$.
- 3) Letting $T \sim_1 T'$ iff $T \leq T'$ and $T' \leq T$ (for T, T' finite or infinite trees).

However, due to the particular form of \sim , both extensions can be shown to be equivalent.

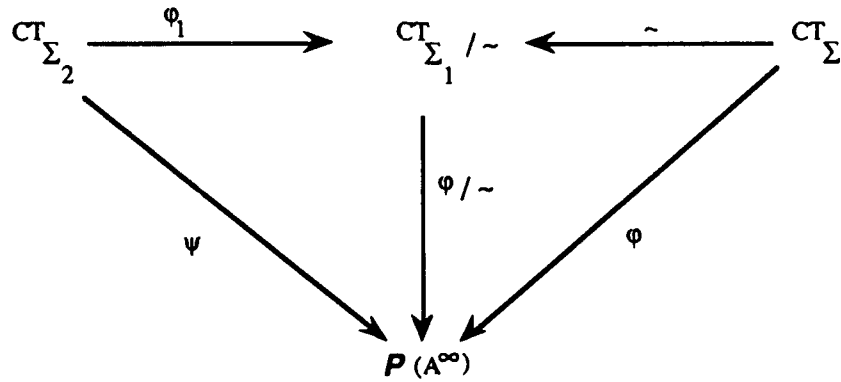


Figure 4.

Recall that CT_{Σ_2} and CT_{Σ_1} now contain an element \perp , representing the undefined process (or computations which are partial but will eventually terminate). \perp is the least element in CT_{Σ_2} , namely $\perp \prec t$, for all t , and CT_{Σ_2} is ordered by the least ordering compatible with the operations in Σ_2 , and having \perp as least element: i.e. $t \prec t'$ iff t' is deduced from t by substituting trees for occurrences of \perp in t , and \perp 's indicate the cutpoints where branches of t can be extended into branches of t' . The relation t is an initial subtree of t' is now replaced by " $t \prec t'$ ". We then need, in order to fully define the diagram in Figure 4, to first redefine $\phi : CT_{\Sigma_1} \rightarrow \mathcal{P}(A^\infty)$ to take into account the undefined element \perp . To this end, it suffices to extend the previously defined ϕ to trees possibly containing \perp by excluding paths ending with \perp from the set of maximal paths; we thus will not account for such paths in $\phi(T)$, intuitively, "only terminated paths count". Moreover, we have the:

LEMMA IV.13 For t, t' trees in CT_{Σ_1} , such that $t \sim t'$, $\phi(t) = \phi(t')$.

Proof. By induction if t and t' are finite. By the definition of \sim and of $br(t)$ for infinite t and t' . \square

ϕ can thus be factored through \sim : for T in CT_{Σ_1}/\sim , $\beta(T) = \phi/\sim(T)$ is defined by:

- 1) taking the set of maximal paths in T , i.e. paths that either are infinite, or are root to leaf paths with a leaf different from \perp .
- 2) erasing labels $+$ and τ along such paths.

We then will associate with each T in CT_{Σ_2} a tree $\phi_1(T)$ in CT_{Σ_1}/\sim ; $\phi_1(T)$ will represent a "normal form" of T after elimination of the δ 's and taking into account the associativity and commutativity of $+$. We finally will apply a slight variation of ϕ to obtain $\psi(T) = \phi(\phi_1(T))$ in $\mathcal{P}(A^\infty)$. This will yield the commutative diagram of Figure 4. We now define formally $\phi_1(T)$ for T in CT_{Σ_2} .

DEFINITION IV.14 (i) For t a finite tree in T_{Σ_2} , $\phi_1(t) \in T_{\Sigma_1}/\sim$ is defined inductively as follows:

- if $t \in CT_{\Sigma_1}$, then $\phi_1(t) = [t]_\sim$,
- if $t = NILot'$, or $t = t'ONIL$, then $\phi_1(t) = \phi_1(t')$,
- if $t = poq$, with $p = \sum_{i \in I} a_i p_i + \perp$, and $q = \sum_{j \in J} a'_j q_j + \perp$, then

$$\phi_1(t) = \sum_k \sum_{i \in I, j \in J} \chi_k(a_i, a'_j) \cdot \phi_1(\delta_k(p) o \delta'_k(q)) + \perp$$

$$\text{where: } \delta_k(p) = \begin{cases} p & \text{if } \nu_k = id \\ p_i & \text{otherwise} \end{cases} \quad \delta'_k(q) = \begin{cases} q & \text{if } \nu'_k = id \\ q_j & \text{otherwise} \end{cases}$$

(The \perp in $\phi_1(t)$ disappears if there is no \perp in neither p nor q),

- if $t = poq$, with p and q not of the above form, then $\phi_1(t) = \phi_1(\phi_1(p) o \phi_1(q))$,
- if $t = \sum_{i \in I} a_i p_i (+ \perp)$, then $\phi_1(t) = \sum_{i \in I} a_i \phi_1(p_i) (+ \perp)$.

(ii) for $T = \sup\{t_n/n \in N\}$ an infinite tree in CT_{Σ_2} , we define $\phi_1(T) = \sup\{\phi_1(t_n)/n \in N\} \in CT_{\Sigma_1}/\sim$.

The Definition IV.14(i) makes sense because $t \sim t'$ implies $\phi_1(t) = \phi_1(t')$, and Definition IV.14(ii) makes sense because CT_{Σ_1}/\sim is complete for the ordering \prec with least element \perp , and because of the following lemma:

LEMMA IV.15 For t_1, t_2 finite trees in CT_{Σ_2} such that $t_1 \prec t_2$, $\phi_1(t_1) \prec \phi_1(t_2)$ in CT_{Σ_1}/\sim .

Proof. By induction on $(d(t_1), \delta(t_1))$, where $d(t)$ is the depth of t , and $\delta(t) = \max\{d(t') + d(t'')/t'ot'' \text{ is a subtree of } t\}$; the pair $(d(t_1), \delta(t_1))$ is ordered lexicographically, and the only non-trivial case is to check the inductive step when $t_1 = poq$. \square

$\phi_1(t)$ can be considered to be an o -free normal form of t in CT_{Σ_1}/\sim .

EXAMPLE IV.16 Let $\Delta = \{a, \bar{a}\}$, $\|\Delta$ be defined as in Example IV.11, $t = aNIL \|\Delta b^2NIL$ and $T = aNIL \|\Delta b\bar{a}b^\omega$; then:

$$\phi_1(T) = ab\bar{a}b^\omega + b(a\bar{a}b^\omega + \bar{a}T_1 + \tau b^\omega),$$

where $T_1 = ab^\omega + bT_1$, i.e. $\phi_1(t)$ is a $\|\Delta$ -free normal form of t in CT_{Σ_1}/\sim .

$\psi : CT_{\Sigma_2} \rightarrow \mathcal{P}(A^\infty)$ can now be defined by:

$$\psi = \beta \circ \phi_1 = (\phi/\sim) \circ \phi_1$$

as announced in Figure 4. We then have to check the adequacy of the above defined ψ . We will first consider the case of finite trees in CT_{Σ_2} , and check that ψ gives us the right semantics.

PROPOSITION IV.17 Let $\psi_1 : T_{\Sigma_2} \rightarrow \mathcal{P}(A^*)$ be defined inductively using the equations (14'). For finite trees in T_{Σ_2} , $\psi_1(t) = \psi(t)$.

Proof. By induction on the number $|t|_o$ of occurrences of o in t .

- If $|t|_o = 0$, then T is in T_{Σ_1} , and $\psi(t) = \phi(t) = \psi_1(t)$ because of the adequacy of the definition of ϕ noted in Section IV.3.
- Assume that for $|t|_o < n$, $\psi(t) = \psi_1(t)$, and let $|t''|_o = n$. We then need an induction on the structure of t'' . All cases are trivial except the case when $t'' = tot'$, with $|t''|_o < n$ and $|t|_o < n$. In that case we obtain by the inductive hypothesis:

$$\psi_1(tot') = \psi_1(t)\bar{o}\psi_1(t')$$

Assume that:

$$\phi_1(t) = [p]_{\sim} = \left[\sum_{i=0}^n a_i p_i (+\perp) \right]_{\sim} \text{ and}$$

$$\phi_1(t') = [p']_{\sim} = \left[\sum_{j=0}^m a'_j p'_j (+\perp) \right]_{\sim}$$

and reason by induction on $s = |p| + |p'|$, where $|p|$ is the maximal length of an action path of $p \in T_{\Sigma_1}$, and is defined by:

$$|p| = \begin{cases} 0 & \text{if } p = \perp \text{ or } p = NIL, \\ 1 + |p'| & \text{if } p = \lambda p', \\ \max(|p'|, |p''|) & \text{if } p = p' + p'' \end{cases}$$

Note that, in the rest of the proof, we will use an abusive notation and identify $\phi_1(t) = [p]_{\sim}$ with its representative p which is an element of the class $\phi_1(t)$, i.e. we write $\phi_1(t) = p$. Noting then that: $\phi_1(t) = \phi_1(\phi_1(t)) = \phi_1(p)$, we obtain:

$$\begin{aligned} \psi_1(t) &= \psi(t) = \beta(\phi_1(t)) = \beta(\phi_1(p)) \\ &= \psi(p) = \psi_1(p) \end{aligned} \quad (19)$$

The first equality follows by induction because $|t|_o < |t''|_o$, and also the last equality, because $|p|_o = 0$. We thus have: $\psi_1(t) = \psi_1(\phi_1(t))$. We now check that: $\psi_1(t'') = \psi(t'')$. If $s = 0$, then $\psi_1(t) = \psi(t) = \varepsilon$. Otherwise:

$$\begin{aligned} \psi(t'') &= \beta(\phi_1(t'')) = \beta(\phi_1(\text{tot}')) = \\ &= \beta(\phi_1(\phi_1(t) \circ \phi_1(t'))) = \\ &= \beta(\phi_1\left[\left(\sum_{i=0}^n a_i p_i + \perp\right) \circ \left(\sum_{j=0}^m a'_j p'_j + \perp\right)\right]) \\ &= \beta\left(\sum_{k,i,j} \chi_k(a_i, a'_j) \phi_1(\delta_k(p) \circ \delta'_k(p'))\right) \\ &= \bigcup_{k,i,j} \chi_k(a_i, a'_j) \beta \circ \phi_1(\delta_k(p) \circ \delta'_k(p')) \end{aligned}$$

Noting that: $\beta \circ \phi_1(\delta_k(p) \circ \delta'_k(p')) = \psi(\delta_k(p) \circ \delta'_k(p'))$ and that $\delta_k(p), \delta'_k(p')$ belong to $\{p, p'\} \cup \{p_i / i = 1, \dots, n\} \cup \{p'_j / j = 1, \dots, m\} \subset T_{\Sigma_1}$, and satisfy $|\delta_k(p)| + |\delta'_k(p')| < |p| + |p'|$, we have by the inductive hypothesis:

$$\beta \circ \phi_1((\delta_k(p) \circ \delta'_k(p'))) = \psi_1(\delta_k(p) \circ \delta'_k(p')).$$

Hence

$$\psi(t'') = \bigcup_{k,i,j} \chi_k(a_i, a'_j) \psi_1(\delta_k(p) \circ \delta'_k(p')).$$

By the definition of ψ_1 , we have: $\psi(t'') = \bigcup_{k,i,j} \chi_k(a_i, a'_j) \psi_1(\delta_k(p)) \bar{\psi}_1(\delta'_k(p'))$ and by "folding" the definition of $\bar{\psi}$, and noting that $\sum_{i=1}^n a_i \psi_1(p_i) = \psi_1(\sum_{i=1}^n a_i p_i)$, we obtain:

$$\psi(t'') = \psi_1(p) \bar{\psi}_1(p').$$

On the other hand:

$$\begin{aligned} \psi(t'') &= \psi_1(t) \bar{\psi}_1(t') \\ &= \psi_1(p) \bar{\psi}_1(p') \end{aligned}$$

by the equation (19).

This proves the proposition. □

We also have the following lemma, which will be quite useful.

LEMMA IV.18 For T, T' in CT_{Σ_1}/\sim :

$$\beta \circ \phi_1(T \circ T') = \bar{\psi}(\beta(T), \beta(T'))$$

Proof. By induction on $s = |t| + |t'|$ for finite t, t' in T_{Σ_1} , and using also the recursive definition IV.10. By the continuity of ϕ_1 for infinite T and T' , using again Definition IV.10. □

COROLLARY IV.19 ψ satisfies the equations (14').

Proof. This is an immediate consequence of Proposition IV.17 for finite trees t, t' in T_{Σ_2} . For infinite trees $T, T' \in T_{\Sigma_2}$, the only case which is not simple is the case of the equality:

$$\psi(ToT') = \bar{o}(\psi(T), \psi(T'))$$

Letting $T = \sup_n t_n$, $T' = \sup_m t'_m$, whence $ToT' = \sup_{(n,m)} t_n o t'_m$, we have:

$$\begin{aligned} \psi(ToT') &= \beta \circ \phi_1[\sup_{n,m} t_n o t'_m] \\ &= \beta[\sup_{n,m} \phi_1(t_n o t'_m)] \\ &= \beta[\sup_{n,m} \phi_1(\phi_1(t_n) o \phi_1(t'_m))] \\ &= \beta o \phi_1[\sup_{n,m} \phi_1(t_n) o \phi_1(t'_m)] \\ &= \beta o \phi_1[\phi_1(T) o \phi_1(T')] \end{aligned}$$

and by Lemma IV.18,

$$\psi(ToT') = \bar{o}[\beta(\phi_1(T)), \beta(\phi_1(T'))] = \bar{o}(\psi(T), \psi(T')).$$

□

V CONCLUSION

We have in the present paper surveyed some applications of algebraic semantics of program schemes:

- an application to logics of programs, through the study of factor algebras of algebras of trees, in particular those corresponding to factors of CT_{Σ} , which characterize the relation \leq_c , defined by $T \leq_c T'$ if and only if $\mathcal{C} \models T \leq T'$; this study was done for classes of interpretations \mathcal{C} which are defined by equations or relations; we gave some hints in Section III.3 on the study of \equiv_c for classes of interpretations \mathcal{C} which are not semi-varieties (\equiv_c is defined by $T \equiv_c T' \iff [T_A = T'_A \quad \forall A \in \mathcal{C}]$). More work remains however to be done for classes of interpretations which are not semi-varieties.
- an application to semantics of concurrency, through the study of transformations of solutions to program schemes, or more generally sets of equations, under factoring morphisms.

VI REFERENCES

- [Aceto-Hennessy] L. Aceto, M. Hennessy, Termination, deadlock and divergence, to appear in *J. Assoc. Comput. Mach.*
- [Andréka-Németi] H. Andréka, I. Németi, Applications of Universal Algebra, Model Theory and Categories in Computer Science, Proc. FCT'81, *Lect. Notes in Comput. Sci.* 117, Springer-Verlag, Berlin (1981), 16-23.
- [ADJ] J. Goguen, J. Thatcher, E. Wagner, J. Wright, Initial Algebra Semantics and Continuous Algebras, *J. Assoc. Comput. Mach.* 24 (1977), 68-95.
- [Arnold-Nivat] A. Arnold, M. Nivat, The metric space of infinite trees: Algebraic and Topological properties, *Fund. Inform.* 3 (1980), 445-476.
- [Bergstra-Klop] J. Bergstra, J. Klop, Algebra of communicating processes, *Proc. CWI Symposium Mathematics and Computer Science*, J. de Bakker, M. Hazenwinkel and J. Lenstra, eds. (1986).
- [Bloom-Tindell] S. Bloom, R. Tindell, Varieties of "IF-THEN-ELSE", *SIAM J. Comput.* 12 (1983), 677-707.
- [Birkhoff] G. Birkhoff, *Lattice Theory*, 3rd edition, *AMS Coll.*, New York (1979).
- [Broy] M. Broy, On the Herbrand-Kleene universe for nondeterministic computations, MFCS 84, LNCS 176, Springer-Verlag (1981), 214-222.
- [Courcelle] B. Courcelle, Arbres infinis et systèmes d'équations, *RAIRO Info. Théor.* 13 (1979), 31-48.
- [Courcelle-Guessarian] B. Courcelle, I. Guessarian, On some classes of interpretations, *J. Comput. and Syst. Sci.* 17 (1978), 388-413.
- [van Emden-Kowalski] M.H. Van Emden, R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Jour. Assoc. Comput. Mach.* 23 (1976), 733-742.
- [Goguen-Meseguer] J. Goguen, J. Meseguer, Initiality, Induction and Computability, in *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds eds., Cambridge Univ. Press (1985), 459-540.
- [Guessarian 89] I. Guessarian, Improving fixpoint tools for computer science, *IFIP'89 Proc.*, G. Ritter ed., North-Holland, Amsterdam (1989), 1109-1114.
- [Guessarian] I. Guessarian, *Algebraic Semantics*, Springer-Verlag, LNCS 99, Berlin (1981).

- [Guessarian-Meseguer] I. Guessarian, J. Meseguer, On the axiomatization of "IF-THEN-ELSE", *SIAM J. Comput.* 16 (1987), 332-357.
- [Grätzer] G. Grätzer, *Universal Algebra*, 2nd edition, Springer-Verlag, Berlin (1979).
- [Mal'cev] A.I. Mal'cev, *Algebraic systems*, North-Holland, Amsterdam (1973).
- [Milner] R. Milner, A calculus of communication systems, *LNCS* 92, Springer-Verlag, Berlin (1980).
- [Nivat] M. Nivat, On the interpretation of recursive polyadic program schemes, *Symp. Mathematica* 15 (1975) 255-281.
- [Rounds] W. Rounds, On the relationships between Scott domains, synchronization trees, and metric spaces, *Inf. and Control* 66 (1985), 6-28.
- [Scott] D. Scott, Data types as lattices, *SIAM Jour. Comput.* 5 (1976), 522-587.

Algebraic Construction of Program Representation Graphs¹

Richard Marciano and Teodor Rus
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242

1 Introduction

Program development environments typically involve operations such as program optimization, program vectorization, program parallelization, program translation, program slicing, program debugging, etc. These operations are performed either by the programmer writing the program or by the compiler translating it. Although all these operations perform the mapping of a given program towards an executable form with some desirable properties, the source and the target of these mappings differ from one another. The techniques to implement them rely on intermediate program representations such as abstract syntax trees, call graphs, def-use chains, control flow graphs, [AhSe88], data dependence graphs [PaKu80] [KuKu81] [PaWo86] [Bran88], program dependence graphs (*PDG*) [FeOt87] [HoRe88], and static single assignment form (*SSA form*) [CyFe89].

The goal of the present research is to develop a formal model, unifying these operations. This model provides a foundation for the construction of a language-based Programmer And Compiler Tool, called *PACT*. *PACT* makes use of an enhanced *PDG* representation of the program called Program Representation Graph, *PRG*. The *PRG* is a graph representation of the source program rather than a graph representation of an intermediate form of the program [AhSe88]. The *PRG* displays both control flow and program dependences, rather than only program dependences as in most *PDG*-like representations [PiBe90]. The main purpose of the *PRG* is to reach a program representation that removes the sequencing constraints imposed by the program's textual layout, explicating the inherent parallelism of a program, thus allowing the mappings performed by the operations involved in program development environments to be naturally integrated by the language design and implementation.

The integration of various program transformations by *PACT* discussed in this paper is based on the development of an algorithm for *PRG* construction which can be embedded within the compiler. In other words, this algorithm allows the construction of a program's *PRG* to be performed by the compiler during program compilation. This is achieved by first constructing a language of *PRGs* called Graph Language, *GL*. Following the algebraic definition of a language [Rus91], the graph language *GL* can be expressed as a triple: $GL = \langle Sem_G, Syn_G, \mathcal{L}_G : Sem_G \rightarrow Syn_G \rangle$, where Sem_G is an algebra representing the semantics of *GL*, Syn_G is an algebra of graphs generated from a given set of nodes and edges, and

¹This work was partially supported by IBM Yorktown, NY, and IBM Rochester, MN.

\mathcal{L}_G a partial mapping constructing for each object of Sem_G a graph representing it (when defined), such that there exists an evaluation homomorphism $\mathcal{E}_G : Syn_G \rightarrow Sem_G$, and for each $a \in Sem_G$, $\mathcal{E}_G(\mathcal{L}_G(a)) = a$, whenever $\mathcal{L}_G(a)$ is defined. For each source language SL processed by the compiler, we provide a mechanism for associating it with a graph language GL . This mechanism consists in choosing the syntax of SL as semantics of GL . This can be done as follows:

1. Let $SL = \langle Sem_S, Syn_S, \mathcal{L}_S : Sem_S \rightarrow Syn_S \rangle$ be a programming language specified by a set P of BNF rules, where Sem_S and Syn_S are similar algebras, and \mathcal{L}_S is a partial mapping between the carriers of Sem_S and Syn_S , such that there exists a homomorphism $\mathcal{E}_S : Syn_S \rightarrow Sem_S$, and for each $a \in Sem_S$, $\mathcal{E}_S(\mathcal{L}_S(a)) = a$, whenever $\mathcal{L}_S(a)$ is defined.
2. The language $GL = \langle Sem_G, Syn_G, \mathcal{L}_G : Sem_G \rightarrow Syn_G \rangle$ associated with SL is constructed as follows:
 - (a) $Sem_G = Syn_S$, i.e., the universe of discourse of the language GL is the collection of syntactic constructs of the language SL .
 - (b) Syn_G is an algebra of graphs. Each graph $G \in Syn_G$ is freely generated by a given finite set of nodes and edges. The nodes of the graph G are labeled by statements and predicates of SL and the edges of the graph G are labeled by symbols denoting control flow, data flow, and dependence information.
 - (c) The mapping $\mathcal{L}_G : Sem_G \rightarrow Syn_G$ is constructed as a homomorphism. For that, we use the fact that Syn_S is a free algebra generated by the signature provided by the collection P of BNF rules specifying the language SL . That is, for each construct $w \in Syn_S$ of the syntactic category A_0 there exist a rule $r \in P$, $r = A_0 \rightarrow t_0 A_1 t_1 \dots t_{n-1} A_n t_n$, and constructs $w_1, w_2, \dots, w_n \in Syn_S$ of the syntactic categories A_1, A_2, \dots, A_n , respectively such that $w = t_0 w_1 t_1 w_2 \dots t_{n-1} w_n t_n$. The homomorphism $\mathcal{L}_G : Syn_S \rightarrow Syn_G$ is defined by associating with each rule $r \in P$, $r = A_0 \rightarrow t_0 A_1 t_1 \dots t_{n-1} A_n t_n$, an operation of graph-construction denoted by $G(r)$ such that $\mathcal{L}_G(w) = G(r)(\mathcal{L}_G(w_1), \mathcal{L}_G(w_2), \dots, \mathcal{L}_G(w_n))$. This is illustrated in Figure 1.

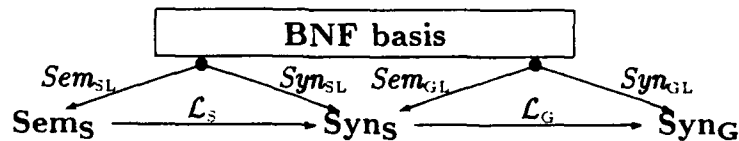


Figure 1: Language-oriented PRG model

Notice that by rule (c) above, Syn_G becomes a graph-algebra having the same signature as the signature of the language syntax algebra Syn_S . In this way the construction of the PRG of a language construct $w \in Syn$ can be performed by an algorithm for computing a homomorphism $T : Syn_S \rightarrow Syn_G$ which is actually the translator part of a compiler $C : SL \rightarrow GL$ [Rus91]. Thus, the algebraic construction of a compiler provides the natural environment for the PACT system.

2 Semantics of GL

The collection of syntactic constructs, Syn_S , of the language SL is a free algebra generated by the signature provided by the given set P of BNF rules in terms of a given set of symbols denoting the lexicon of SL . This property ensures us that the function $\mathcal{L}_G : Sem_G \rightarrow Syn_G$ is total and computable [Rus91]. That is, for each well constructed word $w \in Syn$, $w = t_0 w_1 t_1 w_2 \dots t_{n-1} w_n t_n$, there exists a BNF rule $r \in P$, $r = A_0 \rightarrow t_0 A_1 t_1 \dots t_{n-1} A_n t_n$, and the well constructed words w_1, w_2, \dots, w_n , such that $w = t_0 w_1 t_1 w_2 \dots t_{n-1} w_n t_n$. On the other hand the graph-construction operation $G(r)$ preserves this “well-constructed” word feature in the algebra Syn_G . That is, $\mathcal{L}_G(w) = G(r)(\mathcal{L}_G(w_1), \mathcal{L}_G(w_2), \dots, \mathcal{L}_G(w_n))$ is a well-constructed graph in Syn_G . In other words, this approach of handling graph representations of programs is a natural way to develop a semantics for PRGs.

Other attempts at developing a semantics for PDGs consistent with the source program semantics have been proposed, where isomorphic PDGs are proven to have the same behavior, or where the translation from source programs to PDG-like trees is discussed, or where graph rewriting semantics for PDGs are proposed, and more recently where semantics that interpret PDGs as data-flow graphs are developed. Our approach differs from all of the above by setting its foundations on a formal algebraic model for programming languages, thus allowing us to naturally connect a source program and its PRG representation through a homomorphism of similar algebras, specified by the mapping $\mathcal{L}_G : Syn_S \rightarrow Syn_G$.

3 Syntax of GL

The syntactic objects of GL are directed graphs whose nodes and edges are labeled by the symbols of the alphabet $\Sigma = \{assign, predicate, entry, exit, junction, true, false, ctl, D_T, D_F, D_{li}, D_{lc}, D_o\}$. These graphs are generated from the primitive graphs in Figure 2 [CoCo77], by the operations $G(r)$ associated with the BNF rules specifying the SL .

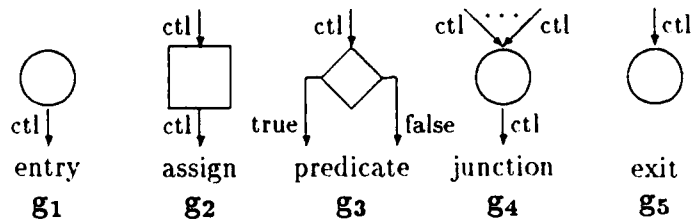


Figure 2: Graph generators

The graph labeling is done during the process of its construction from its components. The initial step of graph construction of $w \in Syn_S$ is the identification and labeling of the primitive graphs composing the graph of w .

Proposition 1: Graph construction is performed by a homomorphism of similar algebras.

Proof: By associating each BNF rule specifying the grammar of the source programming language SL with a graph construction operation of same signature, the graph algebra can be seen as the homomorphic image of the word algebra Syn_S , as expressed by the mapping $\mathcal{L}_G : Syn_S \rightarrow Syn_G$. \square

Proposition 2: Graph construction can be carried out by the algorithm performed by an algebraic compiler.

Proof: The algorithm performed by an algebraic compiler discovers patterns of the form $w = t_0 w_1 t_1 w_2 \dots t_{n-1} w_n t_n$ in the text of the source program, and reduces the text to the left hand-side of the BNF rule $r \in P$ specifying w . During this text reduction, target code is generated by expanding the macro-operation associated with r . If the rule describing the target language of the text specified by r is a graph rule, then the final code generated is the PRG graph of the construct w discovered by the compiler. \square

Proposition 3: Let $SL = \langle Sem_S, Syn_S, \mathcal{L}_S : Sem_S \rightarrow Syn_S \rangle$ be the source language of a compiler and let $GL = \langle Syn_S, Syn_G, \mathcal{L}_G : Syn_S \rightarrow Syn_G \rangle$ be the graph language attached to the source language SL . Then there exists a graph evaluation homomorphism $\mathcal{E}' : Syn_G \rightarrow Sem_S$ such that for all $w \in Syn_S$ and $g \in Syn_G$, if there exists $a \in Sem_S$ and $\mathcal{L}_S(a) = w$, $\mathcal{L}_G(w) = g$, then $\mathcal{E}'(g) = a$.

Proof: Since GL is the graph language attached to SL , then by language definition there exists a homomorphism of similar algebras $\mathcal{E}_G : Syn_G \rightarrow Syn_S$. For the same reason there exists a homomorphism of similar algebras $\mathcal{E}_S : Syn_G \rightarrow Sem_S$. But, since Syn_S and Syn_G are freely generated, $\mathcal{E}_G : Syn_G \rightarrow Syn_S$ can be constructed such that it is an inverse of the homomorphism $\mathcal{L}_G : Syn_S \rightarrow Syn_G$. This means that $\forall g \in Syn_G$, $\mathcal{E}_S(\mathcal{E}_G(g)) \in Sem_S$. That is, $\mathcal{E}' = \mathcal{E}_S \circ \mathcal{E}_G : Syn_G \rightarrow Sem_S$. This is illustrated in Figure 3. \square

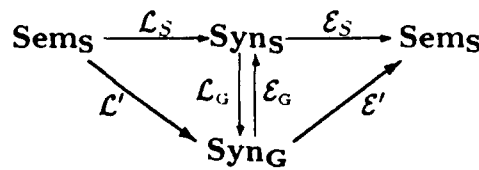


Figure 3: Graph evaluation

Corollary: $GL' = \langle Sem_S, Syn_G, \mathcal{L}_G \circ \mathcal{L}_S : Sem_S \rightarrow Syn_G \rangle$ is a graph language having as the semantics the semantics Sem_S of the source language.

Proof: The corollary results from the algebraic definition of a language and from the construction of the graph language GL . This corollary provides the algebraic model for abstract interpretations [CoCo77] and allows the development of an algebraic algorithm for computing abstract interpretations.

References

- [AhSe88] Aho, A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass. 1986.
- [Bran88] Brandes, T., The importance of direct dependences for automatic parallelization. *Proceedings of the ACM 1988 International Conference on Supercomputing*, July 1988, pp. 407-417.
- [CoCo77] Cousot, P., Cousot, R., Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, (Jan. 1977), pp.238-252.
- [CyFe89] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, (Jan. 1989), pp.25-35.
- [FeOt87] Ferrante, J., Ottenstein, K., Warren, J., The program dependence graph and its use in optimization, *ACM Trans. Program. Lang. Syst.* 9,3 (July 1987),319-349.
- [HoRe88] Horwitz, S., Reps, T., Binkley, D., Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, Ga., June 22-24, 1988), *ACM SIGPLAN Not.*23, 7 (July 1988), 35-46.
- [KuKu81] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., Wolfe, M., Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL)* (Williamsburg, Va., Jan. 26-28). 1981, pp. 207-218.
- [PaKu80] Padua, D. A., Kuck, D. J., Lawrie, D. H., High-Speed Multiprocessors and Compilation Techniques, Special issue on parallel processing, *IEEE Trans. on Computers*, Vol. C-29, No. 9, Sept. 1980, pp.763-776.
- [PaWo86] Padua, D. A., Wolfe, M. J., Advanced compiler optimizations for supercomputers. *Comm. ACM*, vol. 29, no. 12, (Dec. 1986), pp. 1184-1201.
- [PiBe90] Pingali, K., Beck, M., Johnson, R., Moudgill, M., Stodghill, P. *Dependence flow graphs: An algebraic approach to program dependencies*. Tech. Rep. TR 90-1152, Comput. Sci., Cornell University, September 1990.
- [Rus91] Rus, T. Algebraic alternative for compiler construction. *Proceedings of The Unified Computation Laboratory*, C.M.I. Rattray & R.G. Clark (editors). The Institute of Mathematics and Its Applications, Oxford University Press, 1991.

Modification Algebras
G. Ramalingam and Thomas Reps
University of Wisconsin – Madison

1. Introduction

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to merge programs by hand. The program-integration algorithm proposed by Horwitz, Prins, and Reps [Horwitz89] - referred to hereafter as the HPR algorithm - provides a way to create a *semantics-based* tool for integrating a base program with two or more variants. The integration algorithm is based on the assumption that any change in the *behavior*, rather than the *text*, of a program variant is significant and must be preserved in the merged program. An integration system based on this algorithm will determine whether the variants incorporate interfering changes, and, if they do not, create an *integrated* program that includes all changes as well as all features of the base program that are preserved in all variants. Various applications have been envisioned for a semantics-based integration tool [Horwitz89].

This paper is a continuation of two earlier studies of the algebraic properties of the program-integration operation. (A simple example of an algebraic property - which a specific integration algorithm might or might not have - is that of associativity. In this context associativity means: "If three variants of a given base are to be integrated by a pair of two-variant integrations, the same result is produced no matter which two variants are integrated first.") One earlier work that studied the algebraic properties of program integration ([Reps90]) formalized the HPR integration algorithm as an operation in a *Brouwerian algebra* [McKinsey46]. Recently, a new algebraic structure in which integration can be formalized, called *fm-algebra*, was introduced [Ramalingam91]. In *fm-algebra*, the notion of integration derives from the concepts of a *program-modification* and an operation for *combining modifications*. Thus, while the earlier work reported in [Reps90] concerned a homogeneous algebra of *programs*, the later approach concerns a heterogeneous algebra of *programs* and *program-modifications*. Both these frameworks are briefly reviewed in Section 2.

The aim of this paper is to develop the algebraic structure of the domain of program-modifications further. In Sections 3 and 4, we explore the possibilities of defining various operators (on the domain of program-modifications) with a simple intuitive meaning. We investigate one particular model of *fm-algebra*, which represents the HPR algorithm, and show how many of these operators may be defined in terms of function composition alone. We study the properties of these operators and use them to show various properties of program-integration.

The potential benefits of the theory outlined are actually three-fold: (1) It allows one to understand the fundamental algebraic properties of integration—laws that express the "essence of integration." Such laws allow one to reason formally about the integration operation; (2) It provides knowledge that is useful for designing alternative integration algorithms whose power and scope are beyond the capabilities of current algorithms; (3) Because such a theory formalizes certain operations that are more primitive than the integration operation, an implementation of these primitive operations can form the basis for a more powerful program-manipulation system than one based on just the integration operation.

2. Previous Work

2.1. The Brouwerian algebraic framework

Reps [Reps90] shows that the set of all programs written in a simple language may be embedded in an algebraic structure called double Brouwerian algebra, and establishes the correspondence between the HPR program-integration algorithm and a ternary operation of this double Brouwerian algebra. We review these concepts below.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant DCR-8552602, by the Defense Advanced Research Projects Agency, by an IBM graduate fellowship, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, and Xerox.

Definition 2.1. A *Brouwerian algebra* [McKinsey46] is an algebra $(P, \sqcup, \sqcap, \div, \top)$ where

- (i) (P, \sqcup, \sqcap) is a lattice (with \sqcup denoting the join operator, and \sqcap denoting the meet operator) with greatest element \top . The corresponding partial order will be denoted by \sqsubseteq .
- (ii) For all a, b , and c in P , $a \div b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$.

It can be shown that a Brouwerian algebra has a least element, given by $\top \div \top$, which will be denoted by \perp .

Definition 2.2. A *double Brouwerian algebra* [McKinsey46] is an algebra $(P, \sqcup, \sqcap, \div, \div, \top)$ where both $(P, \sqcup, \sqcap, \div, \top)$ and $(P, \sqcap, \sqcup, \div, \top \div \top)$ are Brouwerian algebras.

Definition 2.3. The *integration operator* of a Brouwerian algebra is the ternary operator defined as follows:

$$a \text{ [base] } b \triangleq (a \div \text{base}) \sqcup (a \sqcap \text{base} \sqcap b) \sqcup (b \div \text{base}).$$

We now introduce the notion of a *partial double Brouwerian algebra*, which will prove to be useful later on.

Definition 2.4. Let (P, \sqsubseteq) be a partially ordered set. A subset X of P is said to be *downwards-closed* if $y \sqsubseteq x$ and $x \in X$ implies that $y \in X$.

Definition 2.5. Let $(P, \sqcup, \sqcap, \div, \div, \top)$ be a double Brouwerian algebra. Let S be a downwards-closed subset of P . Then, S is closed with respect to the operations \sqcap and \div , but is not necessarily closed with respect to the operations \sqcup or \div . We call $(S, \sqcup, \sqcap, \div, \div)$ a *partial double Brouwerian algebra*, where \sqcup and \div are partial functions on S , and \sqcap and \div are total functions on S .

Note that a partial double Brouwerian algebra satisfies the various algebraic properties of double Brouwerian algebra, as long as all the relevant expressions are defined.

2.2. The *fm*-algebraic framework

The goal of program-integration is to merge or combine changes made to some program. The concept of a "change made to a program" or a *program-modification* was formalised in [Ramalingam91] and made use of in providing an alternative definition of integration. We briefly review this formalism.

Definition 2.6. A *modification algebra* is a heterogeneous algebra $(P, M, \Delta, +, \langle \rangle)$ with two sets P and M and three operations Δ , $+$, and $\langle \rangle$, with the following functionalities:

$$\begin{aligned} \Delta &: P \times P \rightarrow M \\ + &: M \times M \rightarrow M \\ \langle \rangle &: M \times P \rightarrow P. \end{aligned}$$

For our purposes, we may interpret the components of a modification algebra as follows. P denotes the set of programs; M denotes a set of allowable program-modification operations; $\Delta(a, \text{base})$ yields the program modification done to *base* to obtain a ; operation $m_1 + m_2$ combines two modifications m_1 and m_2 to give a new modification; $m \langle \text{base} \rangle$ denotes the program obtained by performing modification m on program *base*.

We now consider a particular kind of modification algebra, in which the modifications (elements of M) happen to be certain functions from P to P , and $\langle \rangle$ is just ordinary function application.

Definition 2.7. A *functional-modification algebra* (abbreviated *fm*-algebra) is an algebra $(P, M, \Delta, +)$ where $M \subseteq P \rightarrow P$, and Δ and $+$ are operations with the following functionalities:

$$\begin{aligned} \Delta &: P \times P \rightarrow M \\ + &: M \times M \rightarrow M. \end{aligned}$$

Definition 2.8. The 2-variant integration operator $\llbracket _ \rrbracket : P \times P \times P \rightarrow P$ of an *fm*-algebra is defined as follows:

$$a \llbracket \text{base} \rrbracket b \triangleq (\Delta(a, \text{base}) + \Delta(b, \text{base})) \langle \text{base} \rangle.$$

Given a double Brouwerian algebra $(P, \sqcup, \sqcap, \div, \div, \top)$, an *fm*-algebra $(P, M, \Delta, +)$ may be defined as in [Ramalingam91] so that the $\llbracket _ \rrbracket$ operator of the *fm*-algebra is the same as the $\llbracket _ \rrbracket$ operator of the Brouwerian algebra. The goal of this paper is an extended study of this particular model. The tools we use are certain other operators that we define on the set of modifications. Sections 3 and 4 give a brief overview of our results.

3. An algebra of modifications

Let $\mathbf{P} = (P, \sqcup, \sqcap, \div, \div, \top)$ be a double Brouwerian algebra. The elements of P will be referred to as *programs*. The set of program-modifications of interest, M , is given by the following definition.

Definition 3.1. $M \triangleq \{ \lambda z. (z \sqcap y) \sqcup x : x, y \in P \}.$

The word *modification* will denote an element of M . We would like to represent the modification $\lambda z. (z \sqcap y) \sqcup x$ by the ordered pair $\langle x, y \rangle$. This representation will be defined soon, and will prove to be of use in developing the algebraic properties of the set of modifications.

Definition 3.2. Let $P = (P, \sqcup, \sqcap, \div, \cdot, \top)$ be a double Brouwerian algebra. \bar{P} denotes its dual $(P, \sqcap, \sqcup, \cdot, \div, \perp)$. $P \times \bar{P}$ denotes the product algebra of P and \bar{P} . Both \bar{P} and $P \times \bar{P}$ are double Brouwerian algebras.

Definition 3.3. S , a subset of $P \times P$, is defined as follows:

$$S \triangleq \{ \langle x, y \rangle \in P \times P : x \sqsubseteq y \}$$

Let S denote the corresponding partial double Brouwerian algebra induced by S (as a set of elements of $P \times \bar{P}$).

Definition 3.4. A bijection ρ between M and S is defined as follows:

$$\rho(\langle x, y \rangle) \triangleq \lambda z. (z \sqcap y) \sqcup x$$

It can be verified that $\rho^{-1}(m) = \langle m(\perp), m(\top) \rangle$. We use the bijection ρ to represent modifications as ordered pairs. We will often abbreviate $\rho(\langle x, y \rangle)$ to $\langle x, y \rangle$. Any reference to a modification $\langle x, y \rangle$ automatically implies that $x \sqsubseteq y$. Since a Brouwerian algebra is a distributive lattice, $\lambda z. (z \sqcap y) \sqcup x = \lambda z. (z \sqcup x) \sqcap y$ whenever $x \sqsubseteq y$. Let \circ denote function composition. Thus, $m_1 \circ m_2$ denotes the function $\lambda x. m_1(m_2(x))$.

Proposition 3.5.

(a) M is closed with respect to \circ . In particular,

$$\rho(\langle x_1, y_1 \rangle) \circ \rho(\langle x_2, y_2 \rangle) = \rho(\langle x_1 \sqcup (x_2 \sqcap y_1), y_1 \sqcap (y_2 \sqcup x_1) \rangle).$$

(b) Let I denote the modification $\rho(\langle \perp, \top \rangle)$. Then, (M, \circ, I) is a monoid: i.e., \circ is associative, and I is the identity with respect to \circ .

Proposition 3.6. (Extended idempotence). $m_1 \circ m_2 \circ m_1 = m_1 \circ m_2$

Corollary 3.7. \circ is idempotent.

Definition 3.8. Define a binary relation \leq on M as follows:

$$m_1 \leq m_2 \iff m_1 \circ m_2 = m_2 = m_2 \circ m_1$$

The relation \leq is intended to capture the notion of subsumption among modifications. Thus, modification m_1 is subsumed by (or "is contained in", "is smaller than") modification m_2 if performing m_1 and m_2 in either order does nothing more than performing m_2 . The following proposition establishes alternative interpretations of this relation, which may also be used as definitions of \leq .

Proposition 3.9.

(a) \leq is a partial order, with I as the least element.

(b) $m_1 \leq m_2$ iff $m_1 \circ m_2 = m_2$ iff $\exists m. m_2 = m_1 \circ m$.

(c) $m_1 \circ m_2 \geq m_1$.

Proposition 3.10. The function ρ (definition 3.4) is a poset isomorphism from (M, \leq) to S .

It follows from the above proposition that M is itself a partial double Brouwerian algebra with respect to the partial order \leq . We denote the corresponding meet and (partial) join operators \sqcap and \sqcup , and the pseudo-difference and (partial) quotient operators \div and \cdot .

Thus, we know that the meet of any two modifications m_1 and m_2 exists. We may think of $m_1 \sqcap m_2$ as the modification that is "common" to the two modifications m_1 and m_2 . However, two modifications may or may not have a least upper bound. We first formalise a notion of *conflict* between modifications, and use that to establish certain results concerning the least upper bound of two modifications.

Definition 3.11. Modifications m_1 and m_2 are said to *conflict* or *interfere* with each other if $m_1 \circ m_2 \neq m_2 \circ m_1$. Otherwise, they are said to *commute* or *be compatible* with each other.

Proposition 3.12

(a) If m_1 and m_2 commute, then $m_1 \sqcup m_2$ exists and equals $m_1 \circ m_2$.

(b) If $m_1 \sqcup m_2$ exists then m_1 and m_2 commute.

(c) m_1 and m_2 commute iff m_1 and m_2 have a least upper bound iff m_1 and m_2 have an upper bound.

(d) If $m_3 \leq m_1$ and $m_4 \leq m_2$ and if m_1 and m_2 commute then m_3 and m_4 commute.

(e) If $m_1 \geq m_2$ then $m \circ m_1 \geq m \circ m_2$.

Definition 3.13. For any a and b in P , let $M_{a,b}$ denote the set of modifications that map b to a . Thus,

$$M_{a,b} \triangleq \{ m \in M : m(b) = a \}.$$

Proposition 3.14. $M_{a,b}$ has a least element with respect to \leq , namely $\langle a \div b, a \div b \rangle$.

Definition 3.15. Let $\Delta(a,b)$ denote the least element (with respect to \leq) of $M_{a,b}$.

$$\Delta(a,b) \triangleq \min(M_{a,b}) = \langle a \div b, a \div b \rangle.$$

The above definition chooses $\Delta(a,b)$ to be the least modification that changes b to a .

Proposition 3.16.

- (a) $\Delta(a,b)(b) = a$. (b) $\Delta(m(a),a) \leq m$. (c) $\Delta(a,a) = I$.

Now we define a binary operation $|_c$ such that $m_1 |_c m_2$ may be interpreted as that part of modification m_1 that is *compatible* with modification m_2 .

Definition 3.17. $m_1 |_c m_2 \triangleq (m_2 \circ m_1) \sqcap m_1$.

Proposition 3.18.

- (a) $m_1 |_c m_2 = \max \{ m \leq m_1 : m \text{ is compatible with } m_2 \}$.
(b) $m_1 |_c m_2$ and $m_2 |_c m_1$ are compatible with each other.

Now we consider an operation $+$ that "combines" modifications.

Definition 3.19. $\langle x_1, y_1 \rangle + \langle x_2, y_2 \rangle \triangleq \langle x_1 \sqcup x_2, (y_1 \sqcap y_2) \sqcup x_1 \sqcup x_2 \rangle$.

Proposition 3.20. $(M, +, I)$ is a join semi-lattice with I as the least element. Thus,

- (a) $m + m = m$ (b) $(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$
(c) $m_1 + m_2 = m_2 + m_1$ (d) $m + I = m = I + m$

Proposition 3.21.

- (a) If m_1 and m_2 commute then $m_1 + m_2 = m_1 \circ m_2 = m_1 \sqcup m_2$.
(b) $(m_1 |_c m_2) \sqcup (m_2 |_c m_1) \leq m_1 + m_2$.

As may be observed from the previous proposition, when m_1 and m_2 commute, $m_1 + m_2$ is the same as $m_1 \sqcup m_2$ and $m_1 \circ m_2$. If m_1 and m_2 conflict, we may think of $m_1 + m_2$ as being obtained by resolving the conflict in favour of the more important modification. Now we define a binary operator $|_o$ so that $m_1 |_o m_2$ represents that part of modification m_1 that is not "overruled" by modification m_2 , and an associated operator \div_o .

Definition 3.22. Define the two binary operators $|_o$ and \div_o as follows:

$$m_1 |_o m_2 \triangleq (m_1 + m_2) \sqcap m_1$$

$$m_1 \div_o m_2 \triangleq (m_1 |_o m_2) \div m_2$$

Proposition 3.23.

- (a) $m_1 |_c m_2 \leq m_1 |_o m_2 \leq m_1$ (b) $m_1 |_o m_2$ and $m_2 |_o m_1$ commute.
(c) $m_1 + m_2 = (m_1 |_o m_2) \sqcup (m_2 |_o m_1)$ (d) $m_1 |_o m_2 = (m_1 \div_o m_2) \sqcup (m_1 \sqcap m_2)$
(e) $(m_1 |_o m_2) |_o m_2 = m_1 |_o m_2$ (f) $m_1 + m_2 = (m_1 \div_o m_2) \sqcup (m_1 \sqcap m_2) \sqcup (m_2 \div_o m_1)$

Definition 3.24. Modification m_1 is said to *override part* of modification m_2 if $m_2 |_o m_1 \neq m_2$.

Proposition 3.25.

- (a) m_1 does not override part of m_2 iff $m_1 + m_2 \geq m_2$.
(b) If $m_1 \leq m_2$ and m overrides part of m_1 then m overrides part of m_2 .

Let $\llbracket _ \rrbracket$ denote the integration operator of the *fm*-algebra $(P, M, \Delta, +)$.

Proposition 3.26.

- (a) $a \llbracket base \rrbracket b = ((\Delta(a, base) |_o \Delta(b, base)) \sqcup (\Delta(b, base) |_o \Delta(a, base))) (base)$.
(b) $a \llbracket base \rrbracket b = ((\Delta(a, base) \div_o \Delta(b, base)) \sqcup (\Delta(a, base) \sqcap \Delta(b, base)) \sqcup (\Delta(b, base) \div_o \Delta(a, base))) (base)$.
(c) $\Delta(a \llbracket base \rrbracket b, base) \leq \Delta(a, base) + \Delta(b, base)$.
(d) $\Delta(a \llbracket base \rrbracket b, a) = \Delta(b, base) \div_o \Delta(a, base)$.
(e) If $\Delta(a, base)$ and $\Delta(b, base)$ commute then $a \llbracket base \rrbracket b = \Delta(a, base)(b) = \Delta(b, base)(a)$.
(f) If $\Delta(a, base)$ and $\Delta(b, base)$ commute then $\Delta(a \llbracket base \rrbracket b, base) = \Delta(a, base) + \Delta(b, base)$.

4. Applications

We look at some applications of the various properties derived above. Note that the following results apply to the particular *fm*-algebra defined in the previous section.

The problem of separating consecutive edits on some program into individual edits on the same program is as follows: Given programs *base*, *a*, and *c*, where *a* was obtained by modifying *base* and *c* was obtained by modifying *a*, we seek a program *b* that includes the second modification but not the first. A solution previously proposed [Reps90] was that the program we seek is a solution *x* to the equation $a \llbracket base \rrbracket x = c$. Another solution, suggested by our interpretation of Δ as capturing "program modifications", is that $(\Delta(c, a))(base)$ is the program we seek. We consider the relationship between these solutions below.

Theorem 4.1. If $\Delta(c,a)$ and $\Delta(a,base)$ do not conflict then the equation $a[[base]]x = c$ has a solution for x .

Theorem 4.2. If the equation $a[[base]]x = c$ has a solution for x then $b = \Delta(c,a)(base)$ is a solution of that equation. Further, b is the "least-modified" solution of that equation in the following sense: if $m(base)$ is any solution to the equation, then $m \geq \Delta(b,base)$. (Or, equivalently, for any solution x of the equation, $\Delta(x,base) \geq \Delta(b,base)$).

An apparent similarity between the concept of program integration and that of pushout in category theory gave rise to the following question: is program integration simply a pushout in an appropriately constructed category? We define below a particular category, one in which the objects denote programs, and arrows denote program-modifications.

Definition 4.3. Define a category C as follows: The set of objects is P . For every $a \in P$, and $m \in M$, there is an arrow $\langle a, m, m(a) \rangle$ with domain a and codomain $m(a)$. Thus, every arrow $\langle a, m, m(a) \rangle$ is associated with a program modification m . And, if no confusion is likely, the arrow will be denoted just m . The operation \circ is defined to be function composition. Thus, $\langle b, m_2, c \rangle \circ \langle a, m_1, b \rangle = \langle a, m_2 \circ m_1, c \rangle$.

Proposition 4.4. The category C is skeletal: no two distinct objects are isomorphic.

Theorem 4.5. Consider the figure 1a in category C . If the pushout of this figure exists, then $\Delta(b,a)$ and $\Delta(c,a)$ commute, and the pushout is as shown in figure 1b.

The above theorem establishes certain similarities between the concepts of program-integration and pushout. However, program integration is a "weaker" concept than pushout in that it is always defined (i.e., for any three programs) unlike the pushout.

REFERENCES

Horwitz89.

Horwitz, S., Prins, J., and Rees, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* 11(3) pp. 345-387 (July 1989).

McKinsey46.

McKinsey, J.C.C. and Tarski, A., "On closed elements in closure algebras," *Annals of Mathematics* 47(1) pp. 122-162 (January 1946).

Ramalingam91.

Ramalingam, G. and Rees, T., "A theory of program modifications," To appear in *Proceedings of the Colloquium on Combining Paradigms for Software Development*, (Brighton, UK, April 8-12, 1991), (1991).

Rees90.

Rees, T., "Algebraic properties of program integration.," in *Proceedings of the Third European Symposium on Programming*, (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science*, Vol. 432, ed. N. Jones, Springer-Verlag, New York, NY (1990).

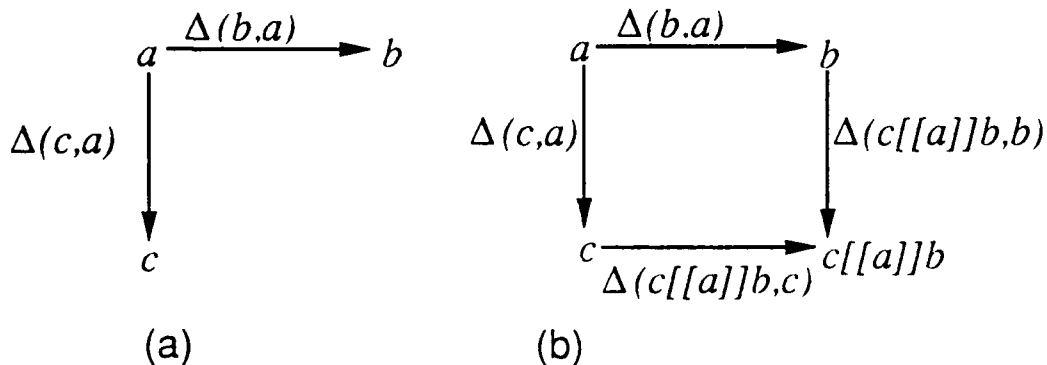


Figure 1. Is integration a pushout?

Decomposition of Finite State Machines under Isomorphic and Bisimulation Equivalences* (Abbreviated Version)

Huajun Qin Philip Lewis

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794, USA
E-mail: qin@sbc.suny.edu

1 Introduction

Almost thirty years after the classic work of Hartmanis and Stearns on the algebraic structure theory of sequential machines[HS66], there is renewed interest in the problem of decomposing a given finite state machine (FSM) into a network of finite state machines that exhibits the same behavior. Often (finite) state machines (or labeled transition systems) are used as an operational semantics of concurrent processes, such as in CCS [Mil80]. Decomposing such a finite state machine into an equivalent network of finite state machines corresponds to a top down design of the corresponding concurrent system.

The finite state machine model that we use is consistent with CCS:

- τ is used to represent an internal action that is invisible externally. When a machine with a τ action is decomposed, the τ action can become an internal communication between the component machines.
- Two new definitions of machine equivalence are used: *strong equivalence* and *observational equivalence* (also called strong bisimulation equivalence and weak bisimulation equivalence) as in CCS. By contrast one definition of equivalence used by Hartmanis and Stearns is *isomorphic equivalence*: the machines are identical except for a renaming of the states.

In general the decomposition problem can be described by the equation

$$X|Y \stackrel{e}{=} M$$

where M is the given specification module, X and Y are the submodules to be determined, $|$ is a composition operator, and $\stackrel{e}{=}$ is an equivalence criterion. Different decomposition methods may be necessary depending on the composition rule, parallelism model (asynchronous or synchronous), and equivalence criterion.

*Partially supported by the National Science Foundation under grant number CCR8822839

In this paper we assume that M , X , and Y are nondeterministic CCS finite state machines and consider four decomposition problems: $X \parallel Y \cong M$, $X \parallel Y \cong M$, $X \parallel Y \sim M$, and $X \parallel Y \approx M$ where \parallel is a synchronous composition operator, \parallel is an asynchronous composition operator, and \cong , \sim , and \approx represent isomorphic, strong, and observational equivalences respectively. (The asynchronous composition operator models the situation in which the two machines have disjoint external action sets and can proceed asynchronously. The two machines may communicate through a set of communication actions and result only τ actions in their composed machine.)

For many equivalence criteria such as strong equivalence, there exist nondeterministic FSMs that are not equivalent to any deterministic FSMs. This makes the study of decomposition of nondeterministic finite state machines essential. In the study of process algebras, the finest equivalence criterion among all the proposed ones is isomorphism. If there is a solution to $X \parallel Y \stackrel{e}{\cong} M$, then there must exist M' such that $M \stackrel{e}{\cong} M'$ and $X \parallel Y \cong M'$ is solvable.

In the following we describe our main results and compare with the related works.

2 Solvability of $X \parallel Y \cong M$

A necessary and sufficient condition for the solvability of $X \parallel Y \cong M$ (a generalization of the Hartmanis and Stearns result to nondeterministic machines) is proved: M is decomposable if and only if there exist two state partitions of M π_1 and π_2 such that

- the pairwise intersection of blocks in π_1 and π_2 results in a trivial partition with each block containing exactly one element;
- for each state s in M if s has x transitions under an action symbol u then blocks (states) B_1 and B_2 that contain s in the quotient machines M_{π_1} and M_{π_2} respectively have y and z transitions under u , and $x = y \times z$.

3 Solvability of $X \parallel Y \cong M$

The problem $X \parallel Y \cong M$ where X and Y have disjoint action sets is considered. A key observation that reveals the relationship between $X \parallel Y \cong M$ and $X \parallel Y \cong M$ is: if we can make our isomorphic decomposition under \parallel in such a way that given an external action u , only one machine makes a transition to a new state while the other machine makes a transition back to the same state (we say that it is "staying at its state"), we can then remove those transitions representing "staying at its state" from the component machines, and thus we obtain an asynchronous isomorphic decomposition under \parallel .

A necessary and sufficient condition for the solvability of $X \parallel Y \cong M$ similar to that of $X \parallel Y \cong M$ is proved. The difference results from the fact that the asynchrony and the communication of two component machines need to be taken into account. This is characterized by a strong confluency property.

4 Solvability of $X \parallel Y \approx M$

The problem $X \parallel Y \approx M$ where X and Y have disjoint action sets is considered. We show that given any nondeterministic machine and any partition of its external actions into two disjoint subsets, we

can always decompose the given machine into two machines with the specified external actions. This is achieved by showing that M can be transformed under observational equivalence to a machine M' and $X \parallel Y \cong M'$ is solvable.

We also show that given an M with e transitions, we can decompose M into a network of e communicating machines with each having 3 states and fewer than 4 transitions. Intuitively each component machine is responsible for simulating one transition in M , and component machines synchronize with each other by 'polling'. This decomposition result is reminiscent of the Krohn and Rhodes prime decomposition theorem [KR62] that any machine can be decomposed into a network of permutation and reset machines under homomorphism.

5 Solvability of $X \parallel Y \sim M$

We have proved a necessary and sufficient condition for the solvability of $X \parallel Y \sim M$ where M is deterministic with respect to non τ actions, and X and Y have disjoint action sets: in M each u -action (with u in the action set of X) should "commute up to \sim " with each v -action (with v in the action set of Y). This is achieved by splitting states to obtain a strongly equivalent M' and $X \parallel Y \cong M'$ is solvable. This property is also the solvability of $X \parallel Y \sim M$ when M is nondeterministic under the restriction that M is acyclic with respect to non τ -transitions. We conjecture that this confluency property is also the solvability condition of $X \parallel Y \sim M$ when M is nondeterministic without the above restriction.

All the proofs of solvability of $X \parallel Y \cong M$, $X \parallel Y \cong M$, $X \parallel Y \approx M$ and $X \parallel Y \sim M$ are constructive. A solution to any of the above decomposition problems can be constructed in polynomial time with respect to the size of M if solutions exist.

6 Related Work

Much research was done on the decomposition theory of sequential machines in the 60's [Har62] [HS62] [HS66] [KR62] [KR65]. There were two main approaches taken: one is the lattice-theoretic by Hartmanis and Stearns, and the other one is the semigroup-theoretic approach by Krohn and Rhodes. Later works such as [Eil76] [Hol82] [Gec86] [DEI89] [MP90] mainly follow the semigroup-theoretic approach. A fundamental result is the Krohn and Rhodes prime decomposition theorem that any sequential machine can be decomposed into a network of permutation and reset machines under homomorphic covering.

Recently Milner and Moller in [Mol89] and [MM90] proved that a finite process (or an acyclic FSM) has a unique decomposition into a network of prime finite processes under strong equivalence and observational congruence. However it was shown that decomposition into a finite set of prime factors does not exist in general for finite state processes.

Other interested works include [Moi85] [BG86] and [Pri85].

References

- [BG86] G. Bochmann and R. Gotzhein. Deriving protocol specifications from service specifications. In *Proceedings of the ACM SIGCOM Symposium*, pages 148–156, 1986.

- [DEI89] P. Dömösi, Z. Ésik, and B. Imreh. On product hierarchies of automata. In *LNCS 380*, pages 137–144, 1989.
- [Eil76] S. Eilenberg. *Automata, Languages and Machines, Vol. B*. Academic Press, New York, 1976.
- [Gec86] F. Gecseg. *Products of Automata*. Springer-Verlag, 1986.
- [Har62] J. Hartmanis. Loop-free structure of sequential machines. *Information and Control*, 5(1):25–43, March 1962.
- [Hol82] W.M.L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982.
- [HS62] J. Hartmanis and R.E. Stearns. Some dangers in state reduction of sequential machines. *Information and Control*, 5(3):252–260, September 1962.
- [HS66] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.
- [KR62] K. Krohn and J. Rhodes. Algebraic theory of machines. In J. Fox, editor, *the Symposium on Mathematical Theory of Automata*, pages 341–384, Polytechnic Institute of Brooklyn, New York, 1962.
- [KR65] K. Krohn and J. Rhodes. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Trans. Amer. Math. Soc.*, 116:450–464, 1965.
- [Mil80] R. Milner. Calculus for communicating systems. In *LNCS 92*, Springer Verlag, 1980.
- [MM90] R. Milner and F. Moller. Unique decomposition of processes. *Bulletin of the EATCS*, 41:226–232, June 1990.
- [Moi85] A. Moitra. Automatic construction of CSP programs from sequential non-deterministic programs. *Science of Computer Programming*, 5:277–307, 1985.
- [Mol89] F. Moller. *Axioms for Concurrency*. PhD thesis, University of Edinburgh, England, July 1989.
- [MP90] O. Maler and A. Pnueli. Tight bounds on the complexity of cascaded decomposition of automata. In *FOCS 90*, pages 672–682, 1990.
- [Pri85] L. Priese. On a fast decomposition method in some models of concurrent computations. *Theoretical Computer Science*, 39:107–201, 1985.

ABSTRACT ALGEBRAIC IMPLEMENTATIONS OF ORDER-SORTED SPECIFICATIONS

V. M. Antimirov, H. V. Melnikova
V. M. Glushkov Institute of Cybernetics
252207, Kiev, prospect Glushkova 20, USSR

The concept of "abstract implementation" of an algebraic specifications $SP1$ by another one $SP2$ is very useful for step-by-step inferential programming. Usually an implementation $SP1 \Rightarrow SP2$ is a syntactic construction equipped with some semantic conditions to distinguish a class of "correct" implementations. The notion of implementation should satisfy the following requirements: 1) methods for proving correctness of an implementation are defined; 2) given two (correct) implementations $SP1 \Rightarrow SP2$ and $SP2 \Rightarrow SP3$ their composition $SP1 \Rightarrow SP3$ is a (correct) implementation; 3) an implementation $SP1 \Rightarrow SP2$ and an enrichment of $SP1$ by new operations to $SP1'$ define new implementation $SP1' \Rightarrow SP2$; 4) an implementation description contains only necessary information (e.g., user need not formulate auxiliary axioms concerned only verification of correctness, as in [1]). We suggest a new concept of abstract implementation for order-sorted algebraic specifications [3], which satisfies the above requirements.

Let $\langle S, \sqsubseteq, \Sigma, E \rangle$ be an order-sorted (o.s.) algebraic specification, where $\langle S, \sqsubseteq, \Sigma \rangle$ is an o.s. regular signature, here on denoted by Σ , and E is a set of Σ -equations. Let specifications $SP_i = \langle \Sigma_i, E_i \rangle$, $i=1,2,3$ be conservative (consistent and sufficiently complete) extensions of the specification $SPO = \langle \Sigma_O, E_O \rangle$. SPO is supposed to be a specification of predefined data types. Let $T_{\Sigma, s}$ denote the set of ground Σ -terms of sort s and \vdash denote the inference relation in order-sorted logic [3].

DEFINITION 1. A *behavioral implementation* of an (abstract) specification $SP1$ by a (resident) one $SP2$ is a triple $IMP12 = \langle \gamma, \phi, SP12 \rangle$, where $SP12 = \langle \Sigma12, E12 \rangle$ is an o.s. algebraic specification, $\gamma: SP1 \dashrightarrow SP12$ is an o.s. signature morphism and $\phi: SP2 \dashrightarrow SP12$ is an o.s. conservative extension, such that:

- 1) γ is identical on Σ_O , i.e., $\gamma(s) = s$ and $\gamma(s) = s$ for each $s \in SO$ $\sigma \in \Sigma_O$;
- 2) for each $s1 \in S1 \setminus SO$ there is a sort $s2 \in S2$ such that $\gamma(s1) \sqsubseteq s2$;
- 3) the implication $E1 \vdash t1 = t2 \Rightarrow E12 \vdash \gamma(t1) = \gamma(t2)$ holds for each sort $s \in SO$ and for each ground terms $t1, t2 \in T_{\Sigma1, s}$.

By this definition two data of $SP1$ may have the same representation in $SP2$ if they are "behaviorally equivalent" [2].

In the following examples the specifications SPO is supposed to contain definitions of natural numbers (sort Nat), positive natural numbers (sort Pnat < Nat) and boolean values (sort Bool) with usual operations.

EXAMPLE 1. Consider $SP1$, the specification of sets of $Pnat$, and $SP2$, the specification of lists and non-empty lists of $Pnat$:

$SP2 = SPO +$

sorts List, Nlist.

subsorts Nlist < List.

op nil:List.

op $_{::}:Pnat, List \rightarrow Nlist$.

$SP1 = SPO +$ **sorts** Set.

op emp:Set.

op $_{+}:Pnat, Set \rightarrow Set$.

vars $x, y:Pnat. S:Set$.

eq $x+x+S=x+S$.

eq $x+y+S=y+x+S$

The implementation $IMP21 = \langle \gamma, \phi, SP21 \rangle$ of $SP2$ by $SP1$ looks as follows: γ, ϕ are imbeddings and

$SP21 = SP1 +$

subsorts List < Set. (i.e. List is implemented by Set)

op nil:List. **op** $_{::}:Pnat, List \rightarrow List$. **vars** $x:Pnat$.

vars $L:List$. **eq** nil=emp. **eq** $x::L=x+L$.

One can prove that the implementation $IMP21$ is behavioral. However if we enrich the $SP2$ to $SP2'$ by some operations such as "head" or "tail", then the correspondent extension of $IMP21$ to the implementation of $SP2'$ by $SP1$ is impossible, because some different lists have the same representation (e.g. $x::y::nil$ and $y::x::nil$). Thus, to satisfy the third requirement mentioned at the beginning it is necessary to make the notion of implementation stricter.

DEFINITION 2. The behavioral implementation $IMP12 = \langle \gamma, \phi, SP12 \rangle$ of $SP1$ by $SP2$ is called *strict implementation* (or, simply, *implementation*) if the condition $E12 \vdash \gamma(t1) = \gamma(t2) \Rightarrow E1 \vdash t1 = t2$ holds for each $s \in S1 \setminus SO, t1, t2 \in T_{\Sigma1, s}$ (*representation consistency*).

Due to the restriction different abstract data have different representations. Then the following fact holds.

THEOREM 1. Let $IMP12 = \langle \gamma, \phi, SP12 \rangle$ is an implementation of $SP1$ by $SP2$ and $SP1' = SP1 + \langle \sigma\Sigma, \sigma E \rangle$ is a conservative enrichment of $SP1$ by operations with resulting sorts from SO . Then $IMP12' = \langle \gamma, \phi, SP12' \rangle$ is an implementation of $SP1'$ by $SP2$, where $SP12' = SP12 + \langle \gamma'(\sigma\Sigma), \gamma'(\sigma E) \rangle$; $\gamma': SP1' \rightarrow SP12', \phi': SP2 \rightarrow SP12'$ are the correspondent extensions of γ, ϕ .

DEFINITION 3. Let $IMP12 = \langle \gamma, \phi, SP12 \rangle$ is a (behavioral) implementation of $SP1$ by $SP2$, $IMP23 = \langle \beta, \psi, SP23 \rangle$ is a (behavioral) implementation of $SP2$ by $SP3$. Then the triple $IMP13 = \langle \alpha \circ \gamma, \xi \circ \psi, SP13 \rangle$ is a *composition* of (behavioral) implementations $IMP12$ and $IMP23$, where $SP13 = SP23 + \langle \Sigma12, \alpha(E12) \rangle$, $\langle \alpha, \xi, SP13 \rangle$ is a pushout of morphisms $\phi: SP2 \rightarrow SP12$ and $\beta: SP2 \rightarrow SP23$.

EXAMPLE 2. Let $SP1'$, $SP2'$ be the following enrichments of $SP1$ and $SP2$ from the example 1, $SP3'$ be the specification of arrays of Pnat:

$SP' = SP1 +$

op $_ \in _ : \text{Pnat}, \text{Set} \rightarrow \text{Bool}$.

op $_ ! : \text{Set} \rightarrow \text{Nat}$.

vars $x, y : \text{Pnat}$. $S : \text{Set}$.

eq $x \in \text{emp} = \text{False}$.

eq $x \in y + S = (x == y) \text{ or } (x \in S)$.

eq $! \text{emp} = 0$.

eq $! x + S = \text{if}(x \in S, !S, !S + 1)$.

$SP2' = SP2 +$

op $\text{hd}(_) : \text{Nlist} \rightarrow \text{Pnat}$.

op $\text{tl}(_) : \text{List} \rightarrow \text{List}$.

op $\text{len}(_) : \text{List} \rightarrow \text{Nat}$.

vars $x : \text{Pnat}$. $L : \text{List}$.

eq $\text{hd}(x :: L) = x$.

eq $\text{tl}(\text{nil}) = \text{nil}$.

eq $\text{tl}(x :: L) = L$.

eq $\text{len}(\text{nil}) = 0$.

eq $\text{len}(x :: L) = \text{len}(L) + 1$.

$SP3' = SPO +$

sorts Array , Pair , Ppair .

subsorts $\text{Pnat} < \text{Nat}$. $\text{Ppair} < \text{Pair}$.

op $\text{null} : \text{Array}$.

op $_ [_] : \text{Array}, \text{Pnat} \rightarrow \text{Nat}$.

op $_ \{ _ : _ \} : \text{Array}, \text{Pnat}, \text{Pnat} \rightarrow \text{Array}$.

op $\text{pr2_} : \text{Pair} \rightarrow \text{Nat}$.

vars $A : \text{Array}$. $x1, x2, k1, k2 : \text{Pnat}$. $i : \text{Nat}$.

eq $\text{pr1} \langle A, i \rangle = A$. **eq** $\text{pr2} \langle A, i \rangle = i$.

eq $A \{ k1 : x1 \} \{ k2 : x2 \} = \text{if}(k1 == k2, A \{ k1 : x2 \}, A \{ k2 : x2 \} \{ k1 : x1 \})$.

eq $A \{ k1 : x1 \} [k2] = \text{if}(k1 == k2, x1, A[k2])$.

eq $\text{null}[k1] = 0$.

op $\langle _, _ \rangle : \text{Array}, \text{Nat} \rightarrow \text{Pair}$.

op $\langle _, _ \rangle : \text{Array}, \text{Pnat} \rightarrow \text{Ppair}$.

op $\text{pr1_} : \text{Pair} \rightarrow \text{Array}$.

op $\text{pr2_} : \text{Ppair} \rightarrow \text{Pnat}$.

The third component of $IMP12'$, implementation of $SP1'$ by $SP2'$, looks as follows:

γ, ϕ are imbeddings and

$SP12' = SP2 +$ **subsorts** $\text{Set} < \text{List}$. (i.e. Set is implemented by List)

op (operations are the same as in $SP1'$)

vars $x, y : \text{pnat}$. $S : \text{Set}$.

eq $\text{emp} = \text{nil}$. **eq** $!S = \text{len}(S)$.

eq $x \in \text{nil} = \text{false}$. **eq** $x + S = \text{if}(x \in S, S, x :: S)$.

eq $x \in y + S = (x == y) \text{ or } (x \in S)$.

For $IMP23'$, the implementation of $SP2'$ by $SP3'$, the third component looks as follows:

$SP23' = SP3' +$ **subsorts** $\text{List} < \text{Pair}$. $\text{Nlist} < \text{Ppair}$.

op (operations are the same as in $SP2'$)

vars $x : \text{Pnat}$. $NL : \text{Nlist}$. $L : \text{List}$.

eq $\text{nil} = \langle \text{null}, 0 \rangle$.

eq $x :: L = \langle (\text{pr1 } L) \{ (\text{pr2 } L) + 1 : x \}, (\text{pr2 } L) + 1 \rangle$.

eq $\text{hd}(NL) = (\text{pr1 } NL) [\text{pr2 } NL]$.

eq $tl(L) = \text{if}((pr2\ L) == O, \text{nil}, < pr1\ L, (pr2\ L) - 1 >)$.
eq $len(L) = pr2\ L$.

One can prove correctness of the above implementations.

To provide correctness of the composition we impose one more requirement.

DEFINITION 4. An implementation $IMP12 = \langle \gamma, \phi, SP12 \rangle$ of $SP1$ by $SP2$ is a *free implementation*, if the condition $E12 \setminus E2 - \gamma(t1) = \gamma(t2) \Rightarrow t1 == t2$ (syntactic equality) holds for each $t1, t2 \in T_{\Sigma1, s'}$ $s' \in S1 \setminus SO$.

Note, if $SP12 \setminus SP2$ has the form of a functional program then $IMP12$ is always free.

THEOREM 2. The composition of two (behavioral) free implementations is a (behavioral) free implementation.

An example of a composition of implementations $SP1' ==> SP2'$ and $SP2' ==> SP3'$ is an implementation of $IMP13'$ with $SP13' = SP23' + \text{subsorts Set} < \text{List} < \text{Pair}$.

op $\{\Sigma12 \setminus \Sigma2\}$. **eq** $\{E12 \setminus E12\}$.

It is easy to check the components of this composition are free implementations, thus it is correct by theorem 2

The crucial point of our approach is the use of subsorts. This allows to excuse user from definition of equality representation [1]. In the full version of the paper some methods for proving correctness of implementations will be described.

References

1. Bernot G. Correctness Proofs for Abstract Implementations.
In: Information and Computation, V.80, 1989, pp. 121-151.
2. Ganzinger H. Parameterized Specification: Parameter Passing and Implementation with Respect to Observability. *Acm Transactions on Programming Languages and Systems*, V.5, No.3, 1983, pp. 318-351.
3. Goguen J., Meseguer J. Order-sorted Algebra 1.- SRI International, Technical Report SRI-CSL-89, July 1989.

Author Index

- V.M. Antimirov, 333
A. Arnold, 45
E. Astesiano, 27
R. Backhouse, 191
M. Baldamus, 101
D. Bernot, 139,163
G. Bert, 287
M. Bidoit, 139
V. Breazu-Tannen, 221
P.J. de Bruin, 191
J. Cai, 185
F. Cornelius, 101
A. Cornell, 97
G. Deutsch, 67
T.B. Dinesh, 41
H. Eertink, 273
H. Ehrig, 101
D. Eichmann, 37
W. Fey, 131
M-C. Gaudel, 163
A. Giovani, 27
I. Guessarian, 297
A. Guha, 231
H. Haughton, 293
G. S. Hirst, 41
P. Hoogendijk, 191
X. Hua, 231
H. Hussmann, 171
P. Inverardi, 87
R. Janicki, 83
N.D. Jones, 245
S. Kaplan, 67
E.W. Karlsen,
J. Knaack, 175
E. Kounalis, 235
M. Koutny, 83
B. Krieg-Brückner, 269
C. Lafontaine, 287
K. Lano, 293
P. Lewis, 329
G. Malcolm, 191
R. Marciano, 317
B. Marre, 163
H.V. Melnikova, 333
F. Morando, 27
M. Nesi, 87
F. Orejas, 101
R.A. Paige, 185
F. Parisi-Presice, 31
V. Pratt, 7
H. Qin, 329
G. Ramalingam, 323
G. Reggio, 27
T. Reps, 323
D. Rus, 277
T. Rus, 175,317
M. Rusinowitch, 235
P-Y. Schobbens, 127
S.F. Smith, 227
Y.V. Srinivas, 281
R. Subrahmanayan, 221
C.L. Talcott, 135
M. Thomas, 241
O. Traynor, 269
E. van der Meulen,181
J. van der Woude, 191
E. Voermans, 191
P. Watson, 241
M. Zamfir-Bleyberg, 91
H. Zhang, 231

Addresses:

V.M. Antimirov
Glushkov Institute of Cybernetics
252207 KIEV
USSR

A. Arnold
Universite Bordeaux I
351 Cours de la Liberation
F-33405 Talence
France

E. Astesiano
University of Genova
Department of Mathematics
Via L.B. Alberti, 4
16132 Genova
Italy

R. Backhouse
Department of Mathematics and
Computing Science
Eindhoven University of Technology
5600 MB Eindhove
The Netherlands

M. Baldamus
Technische Universitaet Berlin
FB 20 Informatik, Sekr. FR 6-1
Franklin Str. 28/29
W 1000 Berlin 10
Germany

G. Bernot
LIENS, UCR CNRS 1327
Ecole Normale Supérieure
45 rue d'Ulm
F - 75230 Paris cedex 05
France

D. Bert
LIFIA, Institut IMAG
av. Felix Viallet 46

F-38031 Grenoble Cedex
France

M. Bidoit
LIENS, CNRS and
Ecole Normale Supérieure
45, Rue d'Ulm
F - 75230 PARIS cedex 05
France

V. Breazu-Tannen
Department of Computer and
Information Science
University of Pennsylvania
200 South 33rd St.
Philadelphia, PA 19104

P.J. de Bruin
Department of Computer Science
Rijksuniversiteit Groningen
9700 AV Groningen
The Netherlands

J. Cai
University of Wisconsin-Madison
Computer Science Department
1210 West Dayton Street
Madison, Wisconsin 53706

F. Cornelius
Technische Universitaet Berlin
FB 20 Informatik, Sekr. FR 6-1
Franklin Str. 28/29
W 1000 Berlin 10
Germany

A. Cornell
Computer Science Department
Brigham Young University
Provo, Utah 48602

G. Deutsch
Department of Computer Science
Bar-Ilan University
52100 Ramat-Gan
Israel

T.B. Dinesh
The University of Iowa
Department of Computer Science
Iowa City, IA 52242

H. Eertink
Tele-Informatics Group
University of Twente
7500 AE Enschede
The Netherlands

H. Ehrig
Technische Universitaet Berlin
FB 20 Informatik, Sekr. FR 6-1
Franklin Str. 28/29
W 1000 Berlin 10
Germany

D. Eichmann
Department of Statistics and
Computer Science
West Virginia University
Morgantown, WV 26506

W. Fey
Technische Universitaet Berlin
FB 20 Informatik, Sekr. FR 6-1
Franklin Str. 28/29
W 1000 Berlin 10
Germany

M-C. Gaudel
Laboratoire de Recherche en Informatique
C.N.R.S. U.A. 410 "Al Khwarizmi"
Universite Paris-Sud - Bat. 490
F - 91405 ORSAY Cedex
France

A. Giovani
University of Genova
Department of Mathematics
Via L.B. Alberti, 4
16132 Genova
Italy

I. Guessarian
Universite de Paris 6
Departement d'Informatique
4 Place Jussieu
75252 Paris, Cedex 05
France

A. Guha
University of Wisconsin-Madison
Computer Science Department
1210 West Dayton Street
Madison, Wisconsin 53706

H. Haughton
Oxford University
Computing Laboratory
11 Keble Rd
Oxford OX1 3QD
United Kingdom

G. S. Hirst
The University of Iowa
Department of Computer Science
Iowa City, IA 52242

P. Hoogendijk
Department of Mathematics and
Computing Science
Eindhoven University of Technology
5600 MB Eindhove
The Netherlands

X. Hua
Department of Computer Science
The University of Iowa
Iowa City, IA 52242

H. Hussmann
Institut fuer Informatik
Technische Universitaet Muenchen
Postfach 702420
W-8000 Muenchen 2
Germany

P. Inverardi
IEI - CNR
Via S. Maria, 46
I - 56126, PISA
Italy

R. Janicki
Department of Computer Science
McMaster University
Hamilton, Ontario
Canada L8S 4K1

N. D. Jones
DIKU
University of Copenhagen
Universitetsparken 1
DK - 2100 Copenhagen 0
Denmark

S. Kaplan
CNR, URA D0 410
Universite de Paris-Sud
Centre D'Orsay
91405 Orsay Cedex
France

E.W. Karlsen
Computer Research International
DK

J. Knaack
Department of Computer Science
The University of Iowa
Iowa City, IA 52242

E. Kounalis
CRIN, BP 239
54506 Vandoeuvre-les-Nancy
France

M. Koutny
Computing Laboratory
The University of Newcastle
Newcastle upon Tyne
NE1 7RU, U.K.

B. Krieg-Bruckner
PROSPECTRA Project
FB 3 Mathematik-Informatik
Universitaet Bremen
Postfach 330440
D-2800 Bremen 33
Germany

C. Lafontaine
Faculte des Sciences Appliquees
Unite d'Informatique
Place Saint-Barbe 2
B-1348 Louvain-Neuve
Belgium

K. Lano
Oxford University
Computing Laboratory
11 Keble Rd
Oxford OX1 3QD
United Kingdom

Philip Lewis
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

G. Malcolm
Computing Laboratory
Programming Research Group
Oxford University
8-11 Keble Road
Oxford OX1 3QD
United Kingdom

R. Marciano
The University of Iowa
Department of Computer Science
Iowa City, IA 52242

B. Marre
Laboratoire de Recherche en Informatique
C.N.R.S. U.A. 410 "Al Khwarizmi"
Universite Paris-Sud - Bat. 490
F - 91405 ORSAY Cedex
France

H.V. Melnikova
Glushkov Institute of Cybernetics
252207 KIEV
USSR

F. Morando
University of Genova
Department of Mathematics
Via L.B. Alberti, 4
16132 Genova
Italy

M. Nesi
Computer Laboratory
University of Cambridge
Cambridge
UK

F. Orejas
Universitat Politecnica de Catalunya
Departament de Llenguatges i
Sistemes Informatics
Edifici FIB
Pau Gargallo, 5
E - 08028 Barcellona
Spain

R.A. Paige
University of Wisconsin-Madison
Computer Science Department
1210 West Dayton Street
Madison, Wisconsin 53706

F. Parisi-Presice
Department of Mathematics
University of Southern California
Los Angeles, CA 90089-1113

V. Pratt
Stanford University
Computer Science Department
Stanford CA 994305

H. Qin
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

G. Ramalingam
Computer Science Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706

G. Reggio
University of Genova
Department of Mathematics
Via L.B. Alberti, 4
16132 Genova
Italy

T. Reps
Computer Science Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706

D. Rus
Department of Computer Science
306 Upson Hall
Cornell University
Ithaca, NY 14850

T. Rus
The University of Iowa
Department of Computer Science
Iowa City, IA 52242

M. Rusinowitch
INRIA Lorraine, BP 101
54600 Villers-les-Nancy
France

P-Y. Schobbens
Universite Catholique de Louvain
Unite d'Informatique
Place Sainte-Barbe, 2
B - 1348 Louvain-la-Neuve
Belgique

S.F. Smith
The Johns Hopkins University
Department of Computer Science
Baltimore, Maryland 21218

Y.V. Srinivas
Information and Computer Science
University of California
Irvine, CA 92717

R. Subrahmanayam
Department of Computer and
Information Science
University of Pennsylvania
200 South 33rd St.
Philadelphia, PA 19104

C.L. Talcott
Stanford University
Department of Computer Science
Stanford, California 94305-2095

M. Thomas
Department of Computer Science
University of Glasgow
Glasgow G12 8QQ
Scotland
U.K.

O. Traynor
University of Missouri
St-Louis
Missouri

E. van der Meulen
CWI
1009 A.B. Amsterdam
The Netherlands

J. van der Woude
CWI
1009 AB Amsterdam
The Netherlands

E. Voermans
Department of Mathematics and
Computing Science
Eindhoven University of Technology
5600 MB Eindhoven
The Netherlands

P. Watson
Department of Computer Science
University of Glasgow
Glasgow G12 8QQ
Scotland
U.K.

M. Zamfir-Bleyberg
Kansas State University
Department of Computing and
Information Sciences
234 Nichols Hall
Manhattan, Kansas 66506-2302

H. Zhang
Department of Computer Science
The University of Iowa
Iowa City, IA 52242